Tampereen yliopisto

Toni-Erik Martin

# ÄLYKIEKKOJÄRJESTELMÄN WEB-HALLINTATYÖKALU

Informaatioteknologian ja viestinnän tiedekunta
Diplomityö
Lokakuu 2019

# TIIVISTELMÄ

Toni-Erik Martin: Älykiekkojärjestelmän web-hallintatyökalu
Diplomityö
Tampereen yliopisto
Tietotekniikka
Lokakuu 2019

---

Reaaliaikaisesti analysoitua dataa tuottavan älykiekkojärjestelmän ottelun päivystysprosessiin liittyy monia ei-triviaaleja vaiheita, jotka saattavat aiheuttaa ongelmia järjestelmän toiminnalle. Monet näistä ongelmatilanteista johtuvat reaalimaailmassa tapahtuvista inhimillisistä virheistä, joista osaan ei voida vaikuttaa automaattisesti nykyisessä järjestelmässä. Tämän työn tarkoituksena oli toteuttaa järjestelmään päivystäjän toimintaa helpottava ja nopeuttava web-pohjainen hallintatyökalu, jonka avulla voidaan korjata ja esittää näitä virheitä, sekä muuttaa järjestelmän tilaa. Päivystäjät tässä tapauksessa olivat samalla järjestelmän kehittäjiä. Toteutusprosessin piti olla nopeaa, sillä hallintatyökalulle toteutetut ominaisuudet haluttiin ottaa käyttöön välittömästi, vaikka siihen tehdyt ominaisuudet eivät olleet täysin valmiita.

Toteutuksen alussa toiminnalliseksi vaatimukseksi työkalulle asetettiin tagimäärittelyn manuaalinen korjaaminen, joka kuvastaa tietoa siitä, että mikä tagi on määritelty millekin pelaajalle, tuomarille, tai kiekolle ottelussa. Älykiekkojärjestelmä pyrkii tekemään määrittelyn automaattisesti, mutta tämä ei aina onnistu. Esimerkkinä tästä voidaan pitää tilannetta, jossa pelaaja pelaa ensimmäistä kertaa joukkueessa, jolloin järjestelmällä ei ole tietoa kyseisen pelaajan tagista. Tagimäärittelyn lisäksi toteutusprosessin edetessä hallintatyökalulle asetettiin uusia vaatimuksia sitä mukaa, kun päivystäjät kokivat tarpeelliseksi tuoda niitä työkalun vastuulle.

Toteutettu hallintatyökalu käsittää järjestelmän back-endiin toteutetun palvelun, sekä sitä käyttävän front-end -sovelluksen. Back-endiin toteutettu palvelu haluttiin pitää suhteellisen yksinkertaisena, jotta hallintatyökalun back-end -palvelu olisi toimintavarma. Sen vastuulle siis jäi muista palveluista datan haku ja sen tarjoileminen front-end -sovellukselle. Siihen toteutettiin rajapinta, joka suunniteltiin hallintatyökalulle spesifiksi rajapinnaksi. Front-end ja back-end keskustelevat HTTP-protokollan ylitse.

Hallintatyökalun toteutusta arvioitiin toiminnallisuuden sekä käytettävyyden näkökulmasta. Arvioinnissa käsiteltiin sitä mahdollisuutta, että front-end -sovellus hakisi itsenäisesti datansa back-endin palveluista. Pohdinnan tuloksena todettiin, että tämä olisi ollut vain vähän suorituskyvyltään tehokkaampi ratkaisu, kuin työssä toteutettu tapa, eikä tämäkään olisi ollut varmaa. Suorituskykyparannus olisi tullut siitä, että RPC-kutsuja tarvittaisiin vähemmän osassa tapauksista. Toisaalta tässä front-end -sovellus olisi tiukasti sidottu back-endin palveluihin, joka ei myöskään olisi ollut skaalautuva. Käytettävyyden näkökulmasta työkalusta jäi uupumaan reaaliaikaisuuden tuomat hyödyt. Arvioinnin perusteella todettiin, että tavoitteisiin päästiin osittain.

Tulevaisuudessa hallintatyökalun rinnalle toteutetaan uusi hallintatyökalu samoilla ominaisuuksilla. Näin päätettiin siksi, koska älykiekko-otteluiden päivystysmalli muuttuu monikerroksiseksi, sekä se tulee käyttöön liigojen omiin organisaatioihin.

Avainsanat: älykiekko, web, työkalu, tagi, HTTP, RPC

# ABSTRACT

Toni-Erik Martin: Web management tool of a smart hockey system
Master of Science Thesis
Tampere University
Information Technology
October 2019

---

A smart hockey system that produces real-time analyzed data consists of many non-trivial phases which may cause problems in terms of functionality of the system. Many of these problematic situations originates from human errors that may occur in real world situations and which the system can't resolve automatically. The purpose of this work was to implement a web-based management tool for system operator that would be used to correct and present these errors and to manage the state of the system. In this case the operators were also system developers. The process needed to be fast paced because the features that was supposed to be implemented were wanted to be deployed as soon as possible regardless of the state of the implementation.

A requirement was set in the beginning of the process that it should be possible to manually correct a tag mapping, that defines which tag is set to which player, referee or puck. The system tries to set the tag mapping automatically, but this operation does not always succeed. An example of an unsuccessful operation is that when a player is playing in a certain team for the first time in which case the system can't deduce the player's tag for the game. In addition to this requirement, there were new requirements set as the implementation advanced when the operators noticed the need to bring new features to the management tool.

The implemented management tool comprises of a service that is implemented in the back-end and a front-end application that uses that service. The implemented back-end service was kept simple in order to achieve operational reliability. The responsibility of the service was to fetch data from other services in the back-end and to serve data to front-end. An interface was implemented in the back-end service which was designed to be specific to the front-end implementation. Front-end and back-end communicates over HTTP-protocol.

The implementation was evaluated in terms of functionality and usability. A possibility was addressed in which the front-end application would fetch data itself from different back-end services. As a result of the evaluation, it was stated that this would have achieved only a minor improvement in performance compared to the implemented method. Moreover, it was not even a definite improvement. The performance improvement would have been resulted from the fact that there was no need for that many RPC-calls in some cases. On the other hand, this would have resulted in tight coupling between back-end services and the implemented front-end application. Also, this would have not been a scalable solution. From the usability point of view the benefits of real-time solutions were not leveraged at all. The goals of this work were partially achieved as a result of evaluation.

It was decided that in the future there will be implemented another version of the management tool that shares the same features as the tool implemented in this work. This is because the operational organization during a smart hockey game will be transitioned to a multi-layered operation that will be used within the leagues' own organizations.

Keywords: smart hockey, web, tool, tag, HTTP, RPC

# SISÄLLYSLUETTELO

# LYHENTEET JA MERKINNÄT

| | |
|---|---|
| API | engl. Application Programming Interface, ohjelmointirajapinta |
| UDP | engl. User Datagram Protocol, tietoliikenneprotokolla |
| TCP | engl. Transmission Control Protocol, tietoliikenneprotokolla |
| RPC | engl. Remote Procedure Call, tietoliikenneprotokolla |
| SOA | engl. Service Oriented Architecture, palvelukeskeinen arkkitehtuuri |
| BLE | engl. Bluetooth Low Energy, likiverkkotekniikka |
| GCP | engl. Google Cloud Platform, pilvipalvelualusta |
| CLR | engl. Common Language Runtime, ajoympäristö |
| DOM | engl. Document Object Model, dokumenttioliomalli |
| MVC | engl. Model-View-Controller, malli-näkymä-käsittelijä |
| MVP | engl. Model-View-Presenter, malli-näkymä-esittäjä |

# 1. JOHDANTO

Viime vuosina data-analytiikka ja sen sovellusalueet ovat kehittyneet huomattavasti laitteistojen, ohjelmistojen ja pilvipalveluiden tuomien uusien mahdollisuuksien takia. Jääkiekkomaailmassa analytiikkaa on hyödynnetty jo vuosikymmenten ajan, mutta data on ollut epätarkkaa sekä sen hankkiminen on ollut työlästä ainakin jääkiekossa [1]. Käytännössä tähän mennessä jääkiekossa datan hankinta on tehty käsipelillä, mutta teknologian kehittymisen myötä datan hankinnan automatisointi herättää kiinnostusta yhä useammassa urheiluorganisaatiossa.

Älykiekkojärjestelmä on Bitwise Oy:n kehittämä reaaliaikainen analytiikkajärjestelmä, joka on osa Wisehockey-tuotetta. Järjestelmä siis tuottaa lähes automaattisesti reaaliaikaisesti analysoitua dataa, johon liittyy monia eri automatisoituja vaiheita, teknologioita ja laitteistoja. Automaattisen reaaliaikaisen datan tuottaminen ja sen tuominen reaaliaikaisesti käyttäjien saataville on haastavaa, jossa manuaaliselta työltä ei voida välttyä, erityisesti laitteiston asennukseen ja käyttöön liittyvissä asioissa. Tämän lisäksi reaaliaikaisuus tuo järjestelmän eri kokonaisuuksiin omia haasteitaan, jotka vaikuttavat koko järjestelmän toimintaan.

Dataa kerättävissä älykiekko-otteluissa valvotaan järjestelmän toimintaa. Valvonta kuuluu erillisten päivystäjien toimenkuvaan, jotka tämän työn kirjoitushetkellä ovat samalla järjestelmän kehittäjiä. Tämän työn tavoitteena on kehittää järjestelmään web-hallintatyökalu, jonka avulla voidaan ratkaista älykiekko-ottelun päivystyksessä syntyneitä ongelmia, joita voi ilmentyä esimerkiksi inhimillisistä virheistä tai järjestelmän toiminnasta. Sen avulla voidaan siis kontrolloida järjestelmän yksittäisiä komponentteja ja muokata niiden tilaa.

Tässä työssä esitellään ensimmäisenä luvussa 2 älykiekkojärjestelmää korkeammalla tasolla. Luvussa 3 käydään läpi työssä toteutetun hallintatyökalun vaatimuksia ja tämän hetkisiä tulevaisuuden tavoitteita. Luvussa 4 esitellään työn kannalta tärkeät teknologiat ja sovelluskehykset. Luvussa 5 käsitellään toteutetun hallintatyökalun toteutusratkaisuja, ja käydään yksityiskohtaisesti läpi kaikki työkalun sisältämät toiminnallisuudet. Luvussa 6 arvioidaan nykyistä toteutusta ja selvitetään tulevaisuuden käyttötarkoituksia ja jatkokehitystä.

# 2. ÄLYKIEKON JA JÄRJESTELMÄN TAUSTAT

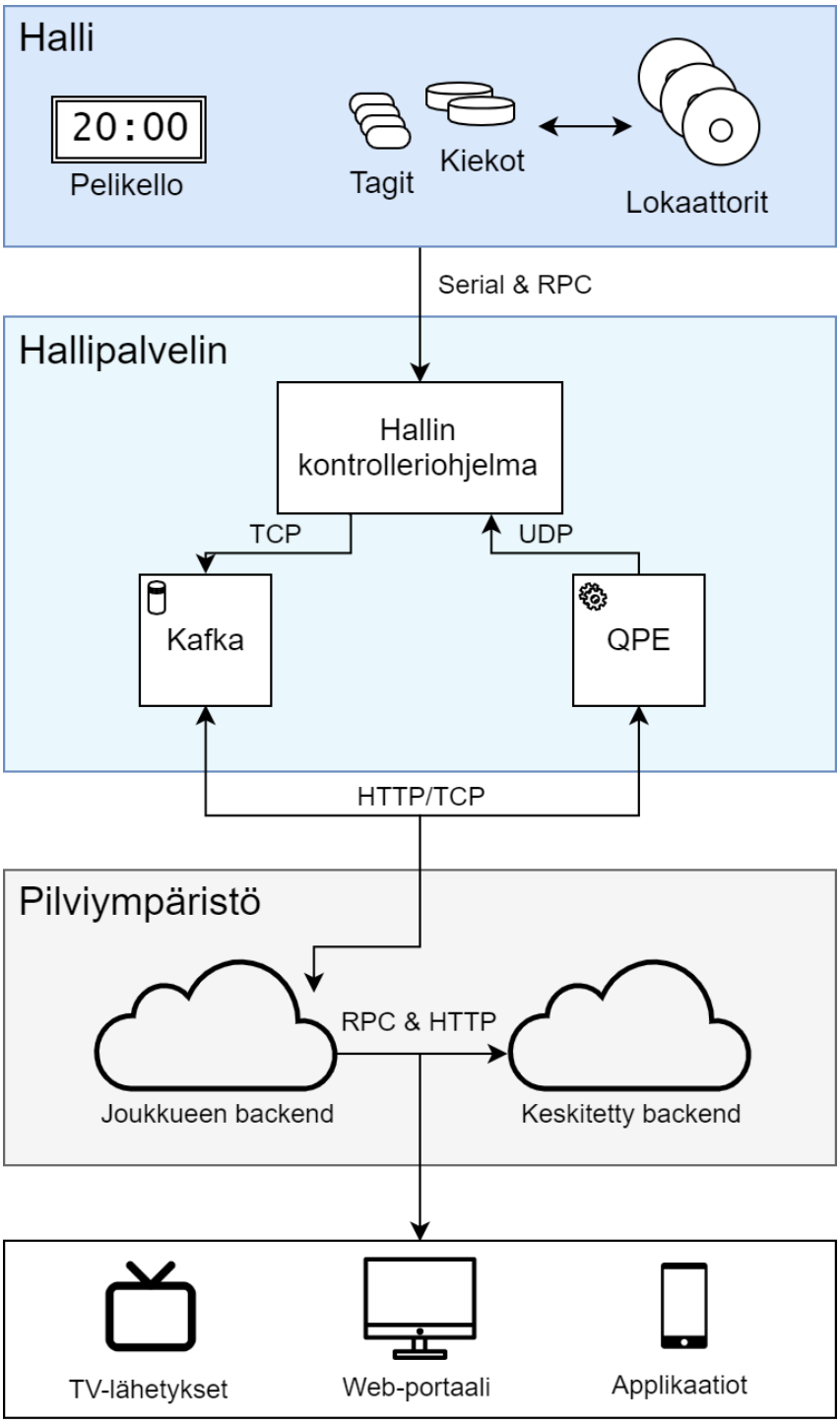
Wisehockey on Bitwise Oy:n lanseeraama sekä kehittämä tuote. Tuotteen keskiössä on järjestelmä, jonka pääasiallisena tavoitteena on tuottaa reaaliaikaisesta paikkadatasta analysoitua dataa ja tuoda se reaaliaikaisesti saataville järjestelmän käyttäjille. Tuotetusta datasta käytetään tämän työn yhteydessä nimitystä *älykiekkodata*. Älykiekkodata sisältää hyvin samankaltaista dataa kuin mitä viralliset jääkiekkoliigat, -seurat, ja -liitot tarjoavat, mutta merkittävänä erona on älykiekkodatan tarkkuus sekä automaattinen datan tuottaminen. Lisäksi älykiekkodatassa on paljon sellaista tietoa, mitä ei voida tuottaa ilman reaaliaikaista paikannusta.

Työn kirjoitushetkellä järjestelmä on ollut käytössä jääkiekkokaudella 2018-2019 Suomen miesten korkeimmalla sarjatasolla, Liigassa [2], viidellä eri joukkueella neljään eri halliin asennettuna. Kyseiset Liigan hallit, joihin järjestelmä on asennettu, sijaitsevat Helsingissä, Tampereella, Turussa, sekä Oulussa. Norjan korkeimmalla sarjatasolla GET-ligaen:ssa [3] järjestelmä on ollut käytössä yhdellä joukkueella keväällä 2019. Tämän lisäksi järjestelmää on demonstroitu Venäjän jääkiekkoliigan KHL:n [4] (Kontinental Hockey League) tähdistöottelussa tammikussa 2019, sekä kolmessa KHL:ssa pelaavan Jokereiden kotiottelussa helmikuussa 2019.

Tuote sisältää älykiekkojärjestelmän lisäksi joukon muita palveluita, jotka yhdessä järjestelmän kanssa muodostavat tuotteen. Tärkeimpinä näistä palveluista voidaan pitää joukkuekohtaista web-portaalia, ottelukohtaista live-videosoitinta, kolmansille osapuolille tarkoitettua API:a, sekä älykiekkodataa sisältävää puhelinsovellusta. Tulevissa aliluvuissa kuvaillaan näitä tuotteen kannalta tärkeitä kokonaisuuksia.

## 2.1 Järjestelmän korkean tason arkkitehtuuri

Älykiekkojärjestelmä voidaan jakaa kahteen eri kokonaisuuteen (kuva 1), joilla on omat päätehtävänsä. Ensimmäinen kokonaisuus koostuu hallin päässä olevista laitteistoista ja *Quuppa Positioning Enginen* (QPE) [5] ympärille rakennetusta halliohjelmistosta, ja toinen kokonaisuus sisältää itse älykiekkojärjestelmän *back-endin*, joka on ajossa pilviympäristössä.




***Kuva 1.*** *Älykiekkojärjestelmä.*

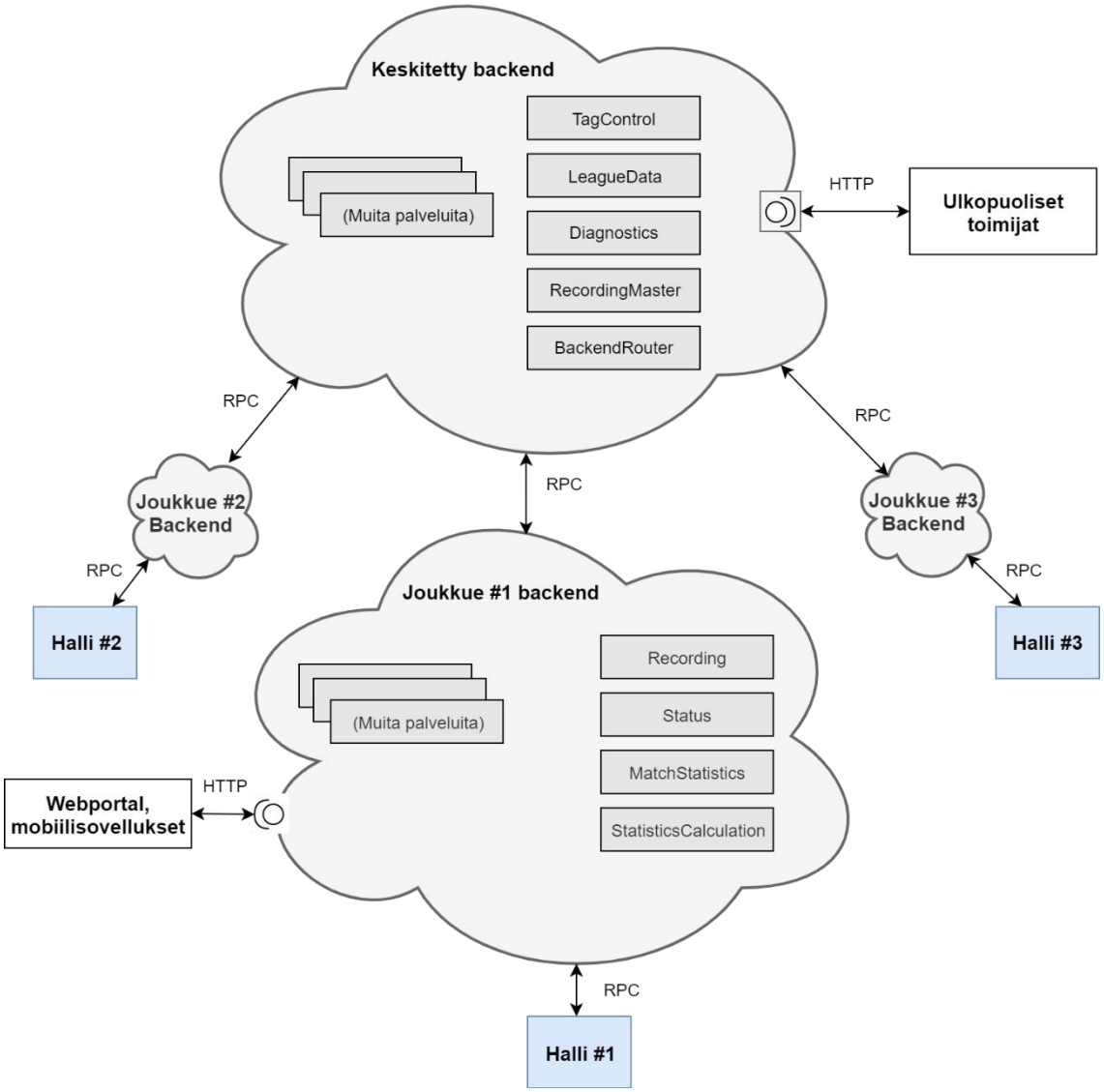
QPE on Quuppa Oy:n [6] toteuttama paikannusjärjestelmä. Sen avulla vastaanotetaan paikkadataa, joka on myös mahdollista syöttää Quupan tarjoamien paikannusalgoritmien läpi tuottaen valmiiksi prosessoitua dataa. Älykiekkojärjestelmän tapauksessa paikkadata kerätään niinsanotusti *raakadatana*, eli sitä ei syötetä minkään Quupan tarjoaman prosessointiputken tai algoritmin läpi, vaan data välitetään sellaisenaan eteenpäin seuraavalle käsittelijälle.

Kuvan 1 hallin kontrolleriohjelma pitää sisällään sovelluksen, joka kuuntelee pelikellolta tulevaa dataa sekä QPE:ltä tulevaa raakapaikkadataa. Raakapaikkadatan siirto tapahtuu UDP-protokollan ylitse, ja kellodatan siirto suoritetaan RPC-kutsulla. Hallin kontrollerisovellus yhdistää paikkadatat kellodataan aikaleimojen avulla, jonka jälkeen se lähettää yhdistetyn datan eteenpäin Kafkaan.

Kafka on suosittu Apachen [7, 8] hajautettu datan säilytykseen ja *suoratoistoon* tarkoitettu alusta, jota käytetään tässä järjestelmässä datan suoratoistoon sekä jossain määrin ottelusta kerätyn datan puskurointiin. Hallin kontrollerisovellus on aina ajossa, joka tarkoittaa sitä, että se lähettää dataa Kafkaan jatkuvasti, vaikka hallissa ei edes olisi jääkiekko-ottelua käynnissä. Kafkassa puskuroidaan dataa viikon ajan, eli dataa ei häviä missään vaiheessa, jos järjestelmä kaatuu tai järjestelmä menettää yhteyden halliin.

Järjestelmän back-end on jaettu niin sanottuun keskitettyyn back-endiin, sekä joukkuekohtaisiin back-endeihin. Keskitettyjä back-endeja on olemassa vain yksi per liiga, joka on yhteinen koko järjestelmälle, kun taas jokaiselle joukkueelle on määritetty oma back-endinsa. Keskitetyn back-endin vastuulla on suorittaa organisoivia toimenpiteitä, jotka ovat luonteeltaan yhteisiä koko järjestelmälle, kuten esimerkiksi liigan tilastojen *pollaus*, ottelun *tallennuksen* käsittely ja ottelun *tagimäärittelyn* luonti. Tagimäärittely tarkoittaa sitä, että jokaiselle pelaajalle, kiekolle, sekä tuomarille on asetettu oma paikkadataa lähettävä Bluetooth-tagi otteluun. Tallenne taasen on järjestelmässä käytetty termi sille, että ottelusta kerätään aktiivisesti dataa pelattavasta ottelusta ja että sitä tallennetaan järjestelmään. Joukkueen back-endin vastuulla on suorittaa joukkuekohtaisia toimenpiteitä, kuten esimerkiksi kommunikointi kotijoukkueen hallin kanssa, älykiekkotilastojen laskenta ja tilastojen tarjoaminen keskitetylle back-endille, sekä kyseisen joukkueen web-portaalin hostaus. Näihin back-endeihin liittyy paljon muitakin asioita, mutta tässä työssä esitetään ja kuvataan vain tämän työn kannalta olennaiset komponentit.

Järjestelmän back-endit noudattavat *mikropalveluarkkitehtuuria*, joka on siis yksi palveluorientoituneen arkkitehtuurimallin (SOA) variaatio. Mikropalveluarkkitehtuurimallilla toteutetut järjestelmät ovat hajautettu itsenäisiksi, löyhästi sidotuiksi, yleensä pienikokoisiksi komponenteiksi, jotka kommunikoivat keskenään rajapintojen kautta. Näitä komponentteja sanotaan mikropalveluiksi. Mikropalveluarkkitehtuurimalli sopii hyvin tämänkaltaisen laajan järjestelmän back-endin toteutukseen, sillä sen pitää kyetä suorittamaan monentyyppisiä toimenpiteitä, joilla voi olla mahdollisesti vaikutusta vain joihinkin järjestelmän mikropalveluihin.




***Kuva 2.*** *Järjestelmän back-endin kokonaisuuksien jaottelu.*

Yllä olevassa kuvassa 2 on esitetty back-endin korkean tason arkkitehtuuri, joissa jokainen back-endin sisällä oleva lohko kuvastaa yhtä mikropalvelua. Lähes jokaisella näistä mikropalveluista on myös oma tietokantansa. Kuvan keskitetystä back-endista sekä joukkueiden back-endeista on jätetty pois sellaiset mikropalvelut, jotka eivät ole tämän työn kannalta oleellisia.

Keskitetyn back-endin *TagControl*-mikropalvelu on vastuussa tagien organisoimisesta. Se pitää kirjaa olemassa olevista tageista ja tagimäärittelyistä, eli mikä tagi on yhdistettynä mihin pelaajaan, kiekkoon, tai tuomariin. Tagit ovat määritelty joukkueittain, eli joukkueet käyttävät yleensä vain oman joukkueen tageja, vaikkakin järjestelmän näkökul-

masta tagimäärittelyssä on mahdollista käyttää muidenkin joukkueiden tageja. Tagimäärittely on ottelukohtainen, sillä tagimäärittely voi vaihtua otteluiden välillä, ja myös ottelukokoonpanot voivat vaihdella otteluiden välillä.

Jääkiekkoliigoilta saatavasta datasta ja sen yhdistämisestä järjestelmään on vastuussa *LeagueData*-mikropalvelu. Se pollaa tietyn väliajoin liigojen tarjoamia rajapintoja, joista saadaan tietoa esimerkiksi joukkueista, pelaajista, otteluista, ottelukokoonpanoista, pistepörsseistä, sekä monista muista yleisistä avoimista tilastoista.

*Diagnostics*-mikropalvelu seuraa tagien toimintaa ottelun aikana suoraan saatavasta raakadatasta. Se tallentaa tiedon siitä, ovatko käytetyt tagit toimintakuntoisia tai onko tagien lähettämässä paikkadatassa suuren aikavälin hyppyjä, eli *vääristymiä*.

*Status*-mikropalvelu vastaa pelikellon tilan ja ottelun tagien reaaliaikaisesta tunnistamisesta. Sen tehtävänä on tunnistaa ottelun aikana Kafkaan tallennetusta datasta kellon tila sekä tagien tila. Tagien tilaan liittyy paikkatiedon lisäksi niiden jännitetiedot. Täten palvelu tunnistaa, mikäli pelikello ja tagit ovat toimintakunnossa. Lisäksi työn kirjoitushetkellä palvelu hostaa hyvin yksinkertaista käyttöliittymää, jonka tehtävänä on näyttää pelikentällä olevan kiekon tila; onko pelissä oleva kiekon tila kelvollinen vai pitääkö kiekko vaihtaa.

*RecordingMaster*-palvelu on keskitetyssä back-endissa oleva palvelu, joka hallinnoi otteluiden tallennuksia ja tekee tarvittavia taustatehtäviä ja käskyttää muita palveluita tarpeen vaatiessa. Esimerkiksi palvelu käskyttää TagControl-palvelua tekemään tagimäärittelyn LeagueData-palvelulta saadulle kokoonpanolle. Lisäksi se käskyttää tiettyjen joukkueiden back-endeja aloittamaan tallennus, mikäli tagimäärittely on tehty onnistuneesti kyseiselle ottelulle.

*BackendRouter*-palvelu pitää kirjaa joukkueiden back-endien *verkkotunnuksista* ja *porteista* jokaisen joukkueen back-endin mikropalveluille. Tätä palvelua käyttämällä keskitetyn back-endin palvelut osaavat yhdistää oikeaan joukkueen back-endiin, mikäli keskitetyn back-endin jokin palvelu tarvitsee tehdä kutsu joukkueen back-endiin.

*Recording*-palvelu on joukkueen back-endissa keskeisessä roolissa. Se muun muassa vastaanottaa Kafkasta hallin dataa ja välittää datan eteenpäin laskentaputkistolle, käskyttää datan arkistoinnin, vastaanottaa RecordingMasterilta tagimäärittelyn, jota tarvitaan tilastolaskennassa, sekä paljon muuta. Palvelun voidaan sanoa toimivan yksittäisen tallenteen hallinnoijana joukkueen back-endissa.

*StatisticsCalculation*-palvelu on vastuussa ottelun älykiekkotilastojen laskennasta. Älykiekkotilastot kirjoitetaan *MatchStatistics*-palveluun, joka on vastuussa tilastojen varastoinnista.

## 2.2 Pelaajien ja kiekon paikannus

Järjestelmän tagien paikannus perustuu Bluetooth Low Energy (BLE) -teknologiaan. Normaaliin Bluetoothiin verrattuna BLE:n tehonkulutus on huomattavasti matalampi kuin Bluetoothilla. Käytännössä tämä tarkoittaa, että BLE:ä sovelletaan tilanteisiin, joissa ei tarvitse siirtää valtavaa määrää dataa. Esimerkiksi videotiedoston siirtäminen laitteelta toiselle on parempi toteuttaa bluetoothilla, kun taas esimerkiksi sykemittari olisi parempi toteuttaa BLE:n avulla pienen datamäärän vuoksi. On arvioitu, että BLE:n virrankulutuksella sitä käyttävät applikaatiot voivat olla yhtäaikaisesti ajossa pienellä paristolla jopa viisi vuotta. [9]

Jokaisessa järjestelmää tukevassa hallissa on asennettuna katossa noin 20 kappaletta Quuppa Oy:n Bluetooth-laitetta, joita kutsutaan *lokaattoreiksi*. Niiden tehtävänä on toimia vastaanottimina tageille. Jokaisen kiekon sisällä ja jokaisen pelaajan paidassa tai kypärässä on Bluetooth-tagi, joilta kyseiset lokaattorit vastaanottavat dataa tietyin aikavälein. Lokaattorit käyttävät Angle-of-Arrival (AoA) signaalinprosessointimenetelmää paikan määrittämiseen, joka perustuu signaalin tulokulmaan sekä aikaerotukseen [10]. Moneen lokaattoriin perustuvalla järjestelyllä pelaajien ja kiekkojen paikka pystytään määrittämään huomattavasti tarkemmin kuin silmämääräisesti voidaan arvioida.

## 2.3 Statistiikka

Älykiekkojärjestelmän keskiössä on tilastolaskentaputkisto, joka on vastuussa älykiekkotilastojen tuottamisesta raakadatan perusteella. Tilastolaskenta on toteutettu joukkueen back-endin StatisticsCalculation-palvelussa. Se koostuu useista rinnakkaisista *datavirta*-lohkoista ja -putkista, joilla jokaisella on omat käyttötarkoituksensa.

Järjestelmässä putket ovat jaettu eri tilastojen kannalta loogisiin osiin siten, että lähes jokaiselle eri tilastolle on oma putkensa. Lisäksi joillekin algoritmeille on määritelty omat suoritusputkensa. Ne voivat olla joko itsenäisiä komponentteja, jotka ovat suorituksessa rinnakkain, tai ne voivat olla riippuvaisia jonkin muun putken tuottamasta tuloksesta. Lisäksi lähes jokaisella näistä on monia eri lohkoja, joilla on omat tehtävänsä. Lohkot voidaan jakaa vastuualueiltaan kolmeen eri lohkotyyppiin. Ensimmäinen tyypeistä on niin sanottu puskurointilohko, jonka tehtävänä on säilöä putkiston dataa datan kuluttajia varten. Kuluttaja tässä tapauksessa tarkoittaa muita putkia. Toinen tyypeistä on suoituslohko, jotka nimensä mukaan suorittavat niille määritelty tehtäviä. Kolmas tyypeistä on ryhmityslohko, joka pääasiassa yhdistävät dataa useasta eri lähteestä. Tilastolaskentaputkisto on hyvin laaja ja monimutkainen kokonaisuus tässä järjestelmässä, eikä sitä

tämän työn puitteissa avata tätä kappaletta enempää. Alla on kuitenkin listattu muutamia sen tuottamia merkittäviä tilastoja:

- Tagin jäälläoloaika ja kuljettu matka aikaleimoineen
- Tagin huippunopeus aikaleimoineen. Tilastossa on mukana myös koordinaatit, missä huippunopeus on mitattu.
- Pelaajavaihdot aikaleimoineen
- Tagin lämpökartta aikaleimoineen. Lämpökartassa kaukalo on jaettu kaksiulotteiseksi matriisiksi, jossa on laskettu, kuinka monta kertaa tagi on näkynyt tietyllä matriisin määrittelemällä alueella.
- Aloitukset aikaleimoineen, joista käy ilmi aloittajien tagit, voittava osapuoli, ja mistä kentän alueelta aloitus on tehty.
- Pelaajan tagin etäisyys kiekkoon aikaleimoineen.
- Laukaukset aikaleimoineen. Tilastossa on mukana tieto siitä, onko jokin pelaaja tai tuomari blokannut laukauksen, onko maalivahti torjunut sen, seurasiko laukauksesta maali vai ei. Mukana on myös laukauksen nopeus sekä laukojan tagi, kiekon tagi, sekä mahdollisen torjujan tai blokkaajan tagi.
- Kiekkokontrollit aikaleimoineen. Tilastossa on siis tieto siitä, mikä tagi kontrolloi kiekkoa, vai onko kiekko niin sanotusti kontrolloimaton. Tilastossa on myös tunnistettu mahdollinen kiekon kamppailutilanne, jossa kiekko ei ole täysin hallussa kellään pelaajalla.
- *Momentum* aikaleimoineen. Tilasto on numeerinen arvo, joka kertoo, kumpi joukkue kontrolloi peliä.

Näiden tilastojen lisäksi putkisto tuottaa vähemmän merkitsevää sisäistä tietoa ja metatietoa, joiden avulla on pyritty helpottamaan back-endin toteutusta, sekä vähentämään back-endissa tehtäviä kutsuja palveluihin.

## 2.4 Muut järjestelmät

### 2.4.1 Web-portaali

Web-portaali on joukkueita varten toteutettu web-käyttöliittymä, joka on web-sovellus älykiekkodatan visualisoinnille. Web-portaali on joukkuekohtainen ja tarkoitettu kyseisen

joukkueen edustajien käyttöön, eli heillä ei ole pääsyä muiden joukkueiden web-portaaleihin. Portaalissa voidaan muun muassa tarkastella monia asioita, joista alla on listattuna merkittävimmät;

- *Laukaisukartta*, jossa esitetään pelaajien laukauksia, joissa on merkittynä laukauksen aikaleima, lähtöpaikka ja nopeus, ja lisäksi tieto siitä päätyikö laukaus maaliin, ohi, vai torjuntaan.
- Pelaajien ja kiekon paikkadatan sekä kiekon hallinnan niin kutsuttu lämpökartta, eli väreillä esitetty kaksidimensionaalinen graafinen esitys datasta kaukalokartassa.
- *Momentum-kaavio*, joka perustuu järjestelmän momentum-statistiikkaan. Kaavio pyrkii kuvaamaan aikajanalla sitä, että kumpi joukkue dominoi peliä.
- *Vaihtojen Gantt-kaavio*, jossa esitetään ottelun kaikkien pelaajien *vaihdot*, eli milloin pelaajat ovat olleet kentällä. Gantt-kaavio on esitystapa, jossa x-akselilla on aikajana, ja vaihtojen kestot esitetään palkkina aikajanalla.

### 2.4.2 Live-videotoistin

Live-videotoistin on sulautettuna web-portaaliin. Se on sovellus, joka visualisoi reaaliajassa järjestelmän tuottamaa paikkadataa. Sovelluksen avulla saadaan kattava yleiskuva koko ottelusta ja sen tapahtumien kulusta. Joukkueet käyttävät sitä erilaisten pelitilanteiden seuraamiseen ottelun aikana, sekä pelien analysointiin ottelun jälkeen. Videotoistin siis tuo uudenlaisen tavan analysoida pelejä ja siten sillä on merkittävä rooli valmennuksessa.

### 2.4.3 API

Järjestelmään on toteutettu kolmansia osapuolia varten rajapinta, josta sitä käyttävät tahot saavat järjestelmän tuottamaan älykiekkodataan.

### 2.4.4 Puhelinsovellukset

Yhdelle Liigan jääkiekkoseuroista, Tapparalle, on toteutettu mobiilisovellus. Se on hyvin tyypillinen faneille suunnattu sovellus, jonka avulla voidaan muun muassa ostaa lippuja otteluun, lukea joukkueen uutisia, tarkastella Liigan tilastoja, tehdä tilauksia hallin ravintoloihin, ynnä muuta. Näiden lisäksi sovelluksessa on älykiekkointegraatio, eli sovelluk-

sessa voidaan tarkastella älykiekkodataa ottelun aikana paikan päällä. Sovellus saa älykiekkodatan järjestelmän API:sta, sekä samasta rajapinnasta mistä web-portaali saa datansa. Sovellus on toteutettu Android- ja iOS-käyttöjärjestelmille.

# 3. HALLINTATYÖKALUN VAATIMUKSET JA TAVOITTEET

Hallintatyökalun pääasiallisena tavoitteena on helpottaa *päivystäjän* toimintaa Liigan ottelupäivänä, sekä tarjota mahdollisuus vaikuttaa järjestelmän tilaan jälkikäteen. Päivystäjä on henkilö, jonka vastuulla on valvoa järjestelmän toimintaa ottelupäivänä, sekä korjata virhetilanteita, joita voi ilmentyä. Ottelupäivän päivystäminen on hyvin virhealtis prosessi, joka pahimmillaan saattaa estää järjestelmän halutun toiminnan. Tässä luvussa kuvataan tarkemmin päivystysprosessia sekä työkalun vaatimuksia ja tavoitteita.

## 3.1 Päivystäjän tehtävien tukeminen

Jokaisessa Liigan ottelussa järjestelmää päivystää siihen dedikoitunut henkilö. Työn kirjoitushetkellä päivystävät henkilöt ovat olleet projektin jäseniä, jotka ovat myös olleet aktiivisesti järjestelmän kehityksessä mukana.

Päivystysprosessi Liigan otteluissa alkaa hetkestä, kun Liiga julkaisee päivystettävien otteluiden kokoonpanot, eli noin kello 13:00. LeagueData-palvelu pollaa Liigan tarjoamasta rajapinnasta alkavien otteluiden kokoonpanoja, jolloin RecordingMaster-palvelu käskyttää TagControl-palvelua tekemään tagimäärittelyn kyseisille kokoonpanoille. Ensisijaisesti TagControl-palvelu yrittää määrittää pelaajille ja tuomareille heidän henkilökohtaisia tagejansa, mutta tämä operaatio ei aina onnistu. Esimerkiksi joukkueen pelaajat voivat vaihtua kesken kauden, jolloin hänelle ei ole määritetty henkilökohtaista tagia järjestelmässä. Pelaajatagit ovat työn kirjoitushetkellä joukkuekohtaisia, eli pääsääntöisesti pelaajien tagimäärittelyissä käytetään kyseisen joukkueen tageja.

### 3.1.1 Toiminnalliset vaatimukset

Toiminnalliset vaatimukset määrittelevät sen, että miten järjestelmän tulisi toimia erilaisissa tilanteissa. Eli ne kuvaavat niitä ominaisuuksia, mitä järjestelmän tulee sisältää, jotta saavutettaisiin järjestelmän haluttu toiminta. [11]

Päivystäjän tehtävään kuuluu korjata tai täydentää vajaavainen tagimäärittely. Hallintatyökalun avulla päivystäjän pitäisi pystyä asettamaan pelaajille ja tuomareille päivystäjän haluama tagi, joka löytyy järjestelmästä, kun päivystäjä saa tiedon oikeasta tagista joukkueen huoltohenkilökunnalta. Joskus tagimuutokset saattavat olla pysyviä, mikäli huoltohenkilökunta näin ilmoittaa. Päivystäjän olisi täten kyettävä asettamaan hallintatyöka-

lulla tagi henkilökohtaiseksi halutuille tuomareille tai pelaajille, sekä tarvittaessa muuttamaan tagin tarkempia tietoja TagControl-palvelussa. Tagimuutokset ja kokoonpanot saattavat myös muuttua, vaikka tagimäärittely olisikin tehty jo aikaisemmin. Tässä tilanteessa työkalulla pitää pystyä tarkistamaan, että ottelun kokoonpano on yhteneväinen tagimäärittelyn kanssa.

Joissakin tilanteissa järjestelmän antamat tilastot voivat olla vääristyneitä. Näin voi tapahtua, jos esimerkiksi jokin tilastolaskennan kannalta oleellinen palvelu on kaatunut tilastolaskennan ollessa käynnissä tai järjestelmään on tehty väärä tagimäärittely, eikä sitä ole korjattu. Luonnollisesti tämä johtaa siihen, että tilastot mahdollisesti lasketaan väärille pelaajille, tuomareille tai kiekoille. Tämän korjaamiseksi hallintatyökalulla pitäisi kyetä aloittamaan tilastolaskenta uudelleen tietylle tallenteelle. Tallenne saattaa olla käynnissä, mikäli hallintatyökalulla muutetaan tagimäärittelyä. Hallintatyökalulla pitäisi myös pystyä uudelleenaloittamaan käynnissä oleva tallennus tällaisessa tilanteessa.

Ottelun aikana kerätään suuri määrä raakadataa kaikista tageista. Hallintatyökalun eräs vaatimus on, että sillä pitää pystyä tarkastelemaan ottelun tagien paikkadatan vääristymiä. Vääristymä tarkoittaa tässä tapauksessa aikaväliä edellisen ja nykyisen paikkadatapaketin välillä. Mitä suurempi aikaväli pakettien välillä on, sitä harvemmin tagi on lähettänyt paikkadataa.

Kiekkotageista tilastolaskentaputkisto kykenee tunnistamaan, mikä on käytössä pelitilanteessa. Käytännössä siis tunnistusalgoritmi tuottaa tiedon, onko kiekko aktiivisena kaukalon rajojen sisällä vai ei. Hallintatyökalun pitäisi tarjota mahdollisuus ylikirjoittaa kiekon tila, jos jostakin syystä algoritmi tunnistaa kiekon tilan väärin.

### 3.1.2 Ei-toiminnalliset vaatimukset

Ei-toiminnalliset vaatimukset kuvaavat sitä, että miten järjestelmän pitäisi toteuttaa toiminnalliset vaatimukset. Siinä siis määritellään järjestelmälle tietyt rajat, joiden sisällä toteutetun järjestelmän on toimittava. Ei-toiminnalliset vaatimukset käsitteenä on itsessään hieman epäselvä, joten ne ovat yleensä jaettu pienempiin alikategorioihin. Nämä voidaan jakaa esimerkiksi seuraaviin kategorioihin [11];

- Kehityksen rajoitteet. Tämä voisi pitää sisällään esimerkiksi hintarajoitteen, kehityksen ehdottoman takarajan, sekä järjestelmän ylläpidettävyyden.
- Arkkitehtuurin rajoitteet. Tämä liittyy suoraan siihen, että miten järjestelmän jakelu on suunniteltu.

- Laadulliset rajoitteet. Tämä kohta sisältää muun muassa käytettävyyden, suorituskyvyn, tietoturvallisuuden sekä asiakaspalvelun.

Hallintatyökalulle asetettiin kehitystyön edetessä laadulliseksi vaatimukseksi se, että sen on oltava käytettävyydeltään sillä tasolla, että lyhyen koulutuksen jälkeen sitä pystyisi operoimaan kuka tahansa liigan organisaatioon kuuluva henkilö. Lisäksi työkalun pitäisi olla niin toimintavarma, että mahdolliset työkalusta koituvat virheet johtuisivat käyttäjän omasta huolimattomuudesta, eikä niinkään huonosta työkalun suunnittelusta. Laadullisten vaatimusten piiriin kuuluu myös työkalun tehokkuus, vaikka sitä vaatimusta ei erikseen asetettu. Tehokkuus tarkoittaa tässä tapauksessa, että sovelluksen käytön tulee olla sujuvaa, vaikka hallintatyökalu tekisi raskaita kutsuja back-endiin.

Arkkitehtuurillisiin rajoitteisiin kuuluu hallintatyökalun tapauksessa se, että sen on oltava web-pohjainen. Näin varmistetaan työkalun alustariippumattomuus. Tämä tarvitsisi Internet-yhteyden, mutta järjestelmän käyttö jo nykyisellään vaatii Internet-yhteyden.

Kehityksen rajoitteiden piiriin kuuluu tässä tapauksessa vain se, että työkalusta on saatava toimiva versio käyttöön niin pian kuin mahdollista. Siinä tulisi olla ensimmäisen käyttöönoton yhteydessä vain tagimäärittelyn mahdollistava ominaisuus.

# 4. TEKNOLOGIAT

## 4.1 Google Cloud Platform -pilvipalveluympäristö

Älykiekkojärjestelmä hyödyntää Googlen pilvipalvelualustaa, Google Cloud Platformia (GCP), joka on käytännössä joukko Googlen tarjoamia resursseja ja palveluita. GCP:n resurssit koostuvat tietokoneista, virtuaalikoneista, kiinto- ja SSD-levyistä (Solid State Drive), sekä yleisistä tietoliikenneteknologioista, jotka sijaitsevat Googlen hajautetuissa datakeskuksissa. Käyttäjät voivat myös rakentaa ja ylläpitää omaa infrastruktuuria, mutta edellä mainitut laitteistot ovat silti käytettävissä virtualisoitujen resurssien muodossa, eli käytännössä virtuaalikoneina pilvessä. Näitä resursseja voidaan hyödyntää GCP:n integroitujen pilvipalveluiden avulla. GCP tarjoaa yli 50 palvelua, jotka noudattavat IaaS-, PaaS-, tai SaaS-pilvipalvelumalleja. Palvelut voidaan jakaa laskentaan, datan varastointiin sekä tietokantoihin, tietokoneverkkoihin, big dataan, koneoppimiseen, tietoturvallisuuteen, sekä hallintatyökaluihin. [12]

Viimeisimmät versiot järjestelmän back-endin palveluista sekä hallintatyökalusta ovat ajossa GCP:n palvelussa nimeltä *Container Engine*. Palvelut voidaan jakaa omiin klustereihinsa Container Enginessä, joilla jokaisella on omat virtuaaliset ytimensä sekä muistinsa. Back-endin palvelut ovat klustereissa *Docker-kontteina*.

GCP:n palvelut noudattavat tyypillisiä pilvipalvelumalleja. Yksi pilvipalvelumalleista on nimeltään *Infastructure as a service* (IaaS), joka nimensä mukaan tarjoaa infastruktuurin käyttäjilleen. Käytännössä se pitää sisällään määritellyt laitteistot ja resurssit, jonka päälle käyttäjä voi rakentaa haluamansa sovellusalustan. *Platform as a service* (PaaS) sisältää samat asiat kuin IaaS, mutta niiden lisäksi PaaS-palvelu yleensä tarjoaa käyttöjärjestelmän, ohjelmointikielikohtaisen ajoympäristön, tietokannat, palvelimet, sekä muut mahdolliset laskenta-alustat. *Software as a service* (SaaS) on PaaS:n päällä oleva taso, joka tarjoaa pääsyn palvelun sovelluksiin yleensä selaimen välityksellä. [13]

## 4.2 Kubernetes & Docker

Docker on joukko PaaS-tuotteita, jotka käyttävät käyttöjärjestelmätason virtualisointia apuna ohjelmiston käyttöönottoprosessissa. Siinä ohjelmistot asetetaan niin kutsuttuihin *kontteihin*, jotka ovat eristettyjä itsenäisiä paketteja omilla konfiguraatioillaan, kirjastoillaan ja ohjelmistoillaan. Sen avulla voidaan eristää sovellukset infastruktuurista, joka mahdollistaa sovelluksien nopean käyttöönoton. Dockerissa sovelluksista ja niiden konfiguraatioista luodaan määrittelyn perusteella kirjoitussuojattu tiedosto, jota kutsutaan

*Imageksi*. Image koostuu useasta tasosta, jossa yleensä Image perustuu toiseen imageen. Esimerkiksi image voi perustua jonkun tietyn käyttöjärjestelmän imageen, mutta sen lisäksi se voi esimerkiksi asentaa palvelimen ja sovelluksen kyseisen imagen määrittelemään ympäristöön. Eli käytännössä image sisältää käännetyn, eli ajettavan version sovelluksesta tietyssä käyttöjärjestelmässä. [14]

Kontteja voidaan ajaa samassa käyttöjärjestelmässä eristettyinä prosesseina, jotka käyttävät samaa prosessoriydintä [14]. Kontit voidaan siis eristää muista konteista ja isäntäkoneesta, joka mahdollistaa konttien ajamisen useassa ympäristössä, vaikka kontin imagea ajettaisiinkin kontin sisällä tietyllä käyttöjärjestelmällä. Kontin konfiguraatiot perustuvat sen imageen sekä mahdollisiin lisäkonfiguraatioihin, mitä käyttäjä asettaa sitä luodessa tai ajettaessa.

Siinä missä Docker luo ja ajaa yksittäisiä kontteja, Kubernetes puolestaan on hallintaalusta, jonka tarkoituksena on *hallinnoida* useita kontteja. Sen avulla voidaan esimerkiksi määritellä miten kontit kommunikoivat keskenään, miten kontteja ajetaan ja millä laitteilla, miten virhetilanteista toivutaan, eli sillä suoritetaan hallinnollisia operaatioita. [15]

Kuberneteksen *pod* on lohko, joka kuvastaa ajossa olevaa prosessia. Käytännössä se on ajossa olevan sovelluksen yksittäinen instanssi, joka voi koostua yhdestä tai useammasta kontista. Jokainen pod toimii kuten looginen kone, eli sillä on oma IP-osoitteensa, isäntänimi, sekä prosessit. Kontit podin sisällä ovat tiukasti sidottuja toisiinsa, jotka jakavat samat resurssit, kuten nimiavaruuden sekä työskentelijän. Älykiekkojärjestelmässä jokaisessa podissa on vain yksi kontti ajossa, eli käytännössä jokainen mikropalvelu on omassa kontissaan ja kuberneteksessä omassa podissa. Poikkeuksena järjestelmässä on palvelut, joita on skaalattu horisontaalisesti, jolloin niistä on useampia instansseja ajossa. [16]

Podit eivät ole pysyviä, vaan niitä käsittelee Kuberneteksen *deploymentit*. Deploymentit voivat luoda, poistaa, tai uudelleenkäynnistää podeja, joka tuo kubernetekseen toimintavarmuutta. Esimerkiksi, jos jokin pod kaatuu, niin deployment pystyy korvaamaan kaatuneen podin identtisellä podilla.

## 4.3 .NET

Järjestelmän back-endin palvelut, mukaan lukien hallintatyökalun back-end -palvelu, pohjautuvat Microsoftin .NET-kehitysalustaan, joka on universaali alusta kaikenlaisten sovellusten kehitykseen, kuten esimerkiksi työpöytäsovellusten, web-sovellusten, sekä IoT-sovellusten kehitykseen. Alustan vahvuutena on tehokas tuki asynkroniseen ja rinnakkaiseen ohjelmointiin, sekä natiivi yhteentoimivuus monen eri alustan kanssa. [17]

.NET-alusta on yksinkertaisuudessaan joukko standardeja, jotka voidaan toteuttaa erilaisille alustoille. Eli tämä tarkoittaa sitä, että .NET-alusta tarjoaa yhtenäiset rajapinnat, joille voidaan tehdä omat implementaationsa. Rajapinnasta ja sen spesifikaatiosta käytetään nimeä .NET Standard. Tarkemmin sanottuna, .NET Standardin rajapintaspesifikaatiot yhdessä muodostavat joukon *sopimuksia* (code contract), joita vastaan sovellukset käännetään, ja nämä sopimukset toteutetaan .NET-implementaatioissa. Tämä mahdollistaa siirrettävyyden eri .NET-implementaatioiden välillä, jotta sovelluksia voidaan ajaa monissa muissakin ympäristöissä. .NET Standard on tarkkaan versioitu, ja se onkin ns. ”target framework”. Se tarkoittaa käytännössä sitä, että jos lähdekoodi (kolmannen osapuolen kirjastot mukaan lukien) on toteutettu tietylle .NET Standardin versiolle, niin sitä voidaan ajaa minkä tahansa .NET implementaation ympäristössä, joka tukee samaa .NET Standard-versiota. [17]

.NET-alustan tärkeimmät implementaatiot ovat nimiltään .NET Framework, .NET Core, Mono, sekä Universal Windows Platform (UWP), jotka ovat myös kaikki Microsoftin ylläpitämiä. Järjestelmässä on käytetty pääosin .NET Corea, sillä se on tarkoitettu järjestelmäriippumattomaksi implementaatioksi .NET-alustasta toisin kuin .NET Framework, joka toteuttaa vain Windows-kohtaiset käyttöjärjestelmän rajapinnat. .NET Core tukee tämän työn kirjoitushetkellä kolmea käyttöjärjestelmää: Linux, Windows, ja OS X. Tämän lisäksi .NET Core on optimoitu pilviympäristöille soveltuviksi, joka soveltuu hyvin tämän järjestelmän käyttöön. .NET Core soveltuu hyvin tämän järjestelmän käyttöön myös jakelun näkökulmasta. Järjestelmän jakelu tapahtuu NuGet-pakettien välityksellä, jossa kehittäjä voi valita käyttöönsä vain tarvitsemansa NuGet-paketit. Tämä soveltuu hyvin mikropalvelujärjestelmiin, sillä kehittäjän ei tarvitse ottaa käyttöön kokonaista monoliittista alustaa. Järjestelmässä on käytetty .NET Standardin versiota 2.0, sekä .NET Coren versiota 2.1. [17]

Jokainen .NET-implementaatio koostuu vähintään kahdesta komponentista: ajoympäristöstä sekä luokkakirjastosta [17]. .NET Corella ajoympäristönä toimii CoreCLR, joka tulee englanninkielen sanoista Core Common Language Runtime. CoreCLR pohjautuu samaan koodiin kuin CLR, joka on .NET Platformille tarkoitettu ajoympäristö. CLR kääntää lähdekoodin ajoaikaisesti JIT-kääntäjällä ja suorittaa käännettyä ohjelmakoodia. Tämän lisäksi se hoitaa muistin käsittelyn automaattisesti, kuten muistin varauksen ja vapauttamisen. CoreCLR suorittaa nämä samat toimenpiteet, mutta se on yksinkertaistetumpi versio CLR:sta, sekä luonnollisesti se on järjestelmäriippumaton ajoympäristö. Luokkakirjasto puolestaan toteuttaa .NET Standardin rajapinnat, sekä näiden lisäksi se tarjoaa tyyppejä ja omia rajapintojaan. Näiden kahden komponentin lisäksi järjestelmässä käy-

tetään kolmatta vaihtoehtoista komponenttia, ASP.NET Corea. Se on web-sovelluskehys, joka tarjoaa työkaluja ja kirjastoja .NET Coren päällä, jotka helpottavat web-sovelluskehitystä.

## 4.4 Angular

Hallintatyökalun käyttöliittymä on toteutettu Googlen ylläpitämällä avoimen lähdekoodin Angular front-end -sovelluskehyksellä. Front-end sovelluskehykset pyrkivät eliminoimaan ongelmia, joita ilmenee tavallisessa pelkkään JavaScriptiin nojaavassa web-kehityksessä. Yksi ehkä isoin ongelma, jonka modernit front-end -sovelluskehykset pyrkivät ratkaisemaan, on käyttöliittymän pitäminen ajan tasalla tilan kanssa. Tämä ei ole kovin työlästä pienemmissä sovelluksissa, mutta vähänkään laajemmissa sovelluksissa tilan hallinta ja käyttöliittymän päivitys voi räjähtää käsiin. Perinteisessä HTML:een ja pelkkään JavaScriptiin tukeutuvissa sovelluksissa tulee monia haasteita vastaan, kun halutaan varmistaa, että koodi on ylläpidettävää, toteutuksessa on noudatettu hyviä suunnitteluperiaatteita, sekä sovelluksen eri komponentit ovat loogisesti jaettu noudattaen DRY-periaatetta (don’t repeat yourself). Siinä pitäisi siis kehittäjän itse toteuttaa ja ylläpitää käyttöliittymän ja bisneslogiikan välillä tapahtumakuuntelijoita ja DOM:n *Query Selector*eita. Front-end -sovelluskehyksillä on jokaisella omanlainen lähestymistapansa, millä näitä ongelmia pyritään ratkaisemaan. [18]

Angular on TypeScriptillä kirjoitettu sovelluskehys, jolla on selkeä lähestymistapa, jonka puitteissa sillä toteutetaan web-sovelluksia. Se perustuu uudelleenkäytettäviin komponentteihin, jotka jokainen pitävät sisällään oman näkymänsä ja siihen liittyvän datatason. Käytännössä komponentti on luokka, joka pitää sisällään sovelluksen datan ja logiikan, ja joka liittyy tiettyyn HTML-templateen, joka määrittelee näkymän. Tämä mahdollistaa löyhän sidonnan komponenttien välillä. Sovelluksen jakaminen omiin komponentteihinsa on loogista myös visuaalisesta näkökulmasta, sillä Internet-sivustot on tyypillisesti jaettu tiettyihin visuaalisiin kokonaisuuksiin. Esimerkiksi tyypillinen uutissivusto pitää sisällään navigointipalkin, uutiset eli sivuston sisällön ja sivuston alatunnisteen. Näiden kaikkien voidaan katsoa olevan omia komponenttejaan toteutuksen näkökulmasta. [18, 19]

Angularissa datan ja näkymän välistä kommunikointia ei tarvitse erikseen toteuttaa, sillä Angular hoitaa sen automaattisesti. Kehittäjän tarvitsee ainoastaan esittää, mikä datalähde liittyy mihinkin näkymän elementtiin. Näiden välisestä yhteydestä käytetään Angularissa nimitystä *kahdensuuntainen sidonta* (data binding). Sidonta voi olla yksi- tai kaksisuuntaista riippuen kehittäjän käyttämästä syntaksista.

Angularissa globaali sovelluslogiikka yleensä toteutetaan erillisiin palveluihin. Esimerkki globaalista logiikasta voisi olla esimerkiksi datan hakeminen palvelimelta. Palvelu on tässä kontekstissa laaja termi, mutta se on käytännössä luokka, jolla on hyvin erityinen käyttötarkoitus, ja joka ei riipu muista komponenteista.

Komponentit voivat käyttää palveluita *riippuvuusinjektion* (dependency injection, DI) avulla. Riippuvuusinjektio on mekanismi, jossa luokka tai objekti pyytää riippuvuutta ulkopuolisesta lähteestä, eli luokan tai objektin ei tarvitse itse luoda riippuvuutta itse. Angularin DI-mekanismi käyttää injektoria, jonka tehtävänä on luoda *singleton*-instanssit palveluista, jotka injektoidaan komponentteihin. Singleton tarkoittaa sitä, että palvelusta on vain yksi instanssi kyseisellä näkyvyysalueella (scope). [17, 18, 20]

## 4.5 PostgreSQL

Monella järjestelmän back-endin mikropalvelulla on oma tietokantansa, jotka ovat tässä tapauksessa kaikki PostgreSQL-tietokantoja. PostgreSQL on avoimen lähdekoodin objektirelationaalinen tietokannanhallintajärjestelmä, joka perustuu Kalifornian yliopiston Berkeleyn Tietotekniikan laitoksen kehittämään POSTGRESiin. Järjestelmä käyttää PostgreSQL:n versiota 9.6. [21]

## 4.6 Kommunikointiprotokollat

### 4.6.1 gRPC

Järjestelmän back-endin palvelut kommunikoivat keskenään käyttäen gRPC-sovelluskehystä. Se on avoimen lähdekoodin järjestelmä etäkäsittelykutsuille (remote procedure call, RPC), joka toimii HTTP:n päällä. RPC toimii konseptina kuten REST-rajapinta, mutta RPC ei käsittele resursseja, vaan sen avulla yleensä suoritetaan toimintoja kutsun vastaanottavissa moduuleissa tai ympäristöissä. RPC käyttää asiakas-palvelin-mallia, jossa asiakas on kutsuja, ja palvelin on toiminnon tarjoaja. gRPC on toteutettu siten, että asiakas voi kutsua palvelimen funktioita aivan kuin ne olisivat kutsujan ympäristössä lokaaleja objekteja [22]. Käytännössä gRPC:n toiminta perustuu käyttäjän määrittelemään rajapintaan, jossa määritellään palvelu ja sen sisältämät funktiot, joita voidaan kutsua määritellyillä parametreilla ja paluuarvoilla. Palvelimen puolella palvelin implementoi rajapinnan ja ajaa gRPC-palvelimen, kun taas asiakaspuolella asiakas luo tyngän, joka tarjoaa samat funktiot kuin palvelin. RPC-kutsut määritellään protocol buffereilla [22], jotka ovat mekanismi strukturoidun datan serialisoimiseksi.

gRPC asiakkaat ja palvelimet voivat kommunikoida keskenään eri ympäristöistä, ja niitä voidaan kirjoittaa eri kielillä. Esimerkiksi palvelin voi olla kirjoitettu C++:lla, kun taas asiakas Pythonilla. [23]

### 4.6.2 HTTP ja REST

Web-hallintatyökalu kommunikoi back-endin kanssa HTTP-protokollan ylitse. Se määrittelee, kuinka WWW:n ylitse lähetetyt viestit ovat rakennettu ja lähetetty, ja mitä toimintoja web-palvelimien ja selainten pitäisi tehdä jokaisella pyynnöllä. HTTP on luonteeltaan tilaton, koska jokainen HTTP-pyyntö suoritetaan riippumatta muista aiemmista tai tulevista HTTP-pyynnöistä. Tämä myös tarkoittaa sitä, että pyynnön mukana käyttäjän on tarjottava kaikki data, mitä palvelin tarvitsee pyynnön suorittamiseksi. Tyypillinen esimerkki HTTP-protokollan käytöstä on, kun käyttäjä navigoi Internet-sivulle verkkoselaimella. Siinä verkkoselain lähettää HTTP GET-pyynnön palvelimelle, johon palvelin vastaa tarjoamalla HTML-sivun käyttäjälle. [24]

Hallintatyökalun back-endissa määritellään rajapinta, joka noudattaa REST-periaatteita. REST tulee sanoista Representational State Transfer, ja se viittaa web-arkkitehtuurityyliin, jolla on oma lähestymistapansa, kuinka hoitaa palvelimen ja asiakkaan välinen kommunikointi. Se on riippuvainen tilattomasta asiakas-palvelin-protokollasta, joka on lähestulkoon aina HTTP-protokolla, kun kyseessä on web-sovellus. REST on suunniteltu siten, että palvelimen objekteja käsitellään resursseina, joita voidaan luoda, muuttaa, tai poistaa. Esimerkki palvelinobjektista voisi olla esimerkiksi käyttäjä, joka tallennetaan järjestelmän tietokantaan; GET-pyynnöllä haetaan käyttäjä, POST-pyynnöllä tallennetaan käyttäjä järjestelmään, ja DELETE-pyynnöllä poistetaan käyttäjä järjestelmästä. REST-rajapinnan yksi merkittävimmistä hyödyistä on se, että käytännössä REST-rajapintoja voi käyttää lähes mikä tahansa asiakasohjelma ohjelmointikielestä riippumatta, sillä REST-rajapinnat käyttävät hyvin tyypillisiä tiedostomuotoja datan välitykseen, kuten esimerkiksi JSON-formaatissa olevia objekteja.

# 5. HALLINTATYÖKALUN TOTEUTUS

Hallintatyökalun toteutus voidaan jakaa kahteen pääkomponenttiin. Ensimmäinen komponentti on back-endiin toteutettu mikropalvelu, joka sijaitsee luvussa 2.1 esitetyssä keskitetyssä back-endissa. Toinen pääkomponentti on Angular-sovellus, joka niin ikään sijaitsee keskitetyssä back-endissa, samassa Docker-kontissa kuin back-endin palvelu. Tässä luvussa käydään läpi hallintatyökalun nykyisen toteutuksen back-endin sekä Angular-sovelluksen toteutusyksityiskohdat.

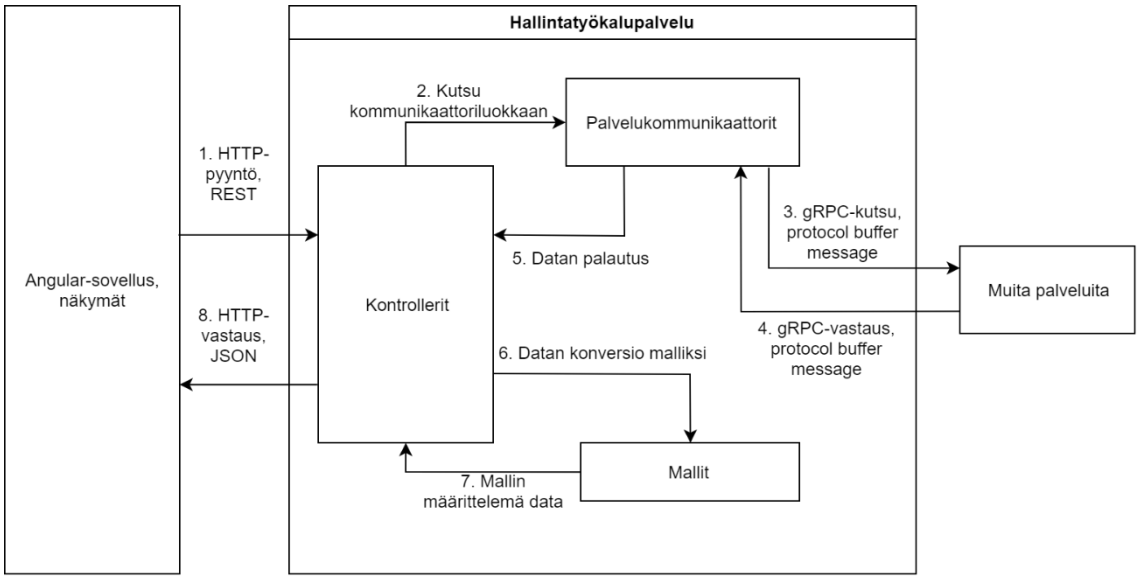
## 5.1 Hallintatyökalupalvelun yleiskuvaus

### 5.1.1 MVC ja Datan kulku

Ensimmäisenä askeleena hallintatyökalun toteutuksessa luotiin itse palvelu järjestelmän keskitettyyn back-endiin. Palvelun tyypiksi valittiin ASP.NET Core, sillä ASP.NET Core sisältää hyödyllisiä web-sovelluksiin tarkoitettuja kirjastoja, joista yksi on ASP.NET Core:n MVC-kirjasto. Sen avulla voidaan luoda helposti MVC-arkkitehtuurimallia noudattava palvelu. MVC-arkkitehtuurimallin kolme kirjainta MVC tulee sanoista malli (model), näkymä (view) ja käsittelijä (controller). Tyypillisessä ASP.NET Core MVC-ohjelmassa ohjelma on jaettu kolmeen osaan, joilla jokaisella osalla on omat vastuualueensa [17]. Siinä mallit määrittelevät sovelluksen dynaamiset tietorakenteet, jotka hallinnoivat ohjelman datarakennetta ja logiikkaa. Näkymät ovat vastuussa mallien tai malleihin liittyvän datan visualisoinneista. Käsittelijät eli kontrollerit vastaanottavat käyttäjän syötteitä ja syöttää ne mallille.

MVC-arkkitehtuurimallin osat ja vastuualueet eivät ole yksiselitteisiä, vaan arkkitehtuurimallista on olemassa monta eri variaatiota riippuen sovelluksesta, sovelluskehyksestä, ja sen käyttötarkoituksesta. Erityisesti web-sovelluksissa, jotka käyttävät MVC-mallia, vastuualueet saattavat olla osittain palvelimen päässä ja osittain asiakkaan päässä. Hallintatyökalussa MVC-mallia on sovellettu siten, että mallit ja kontrollerilogiikat ovat palvelimen päässä, kun taas kaikki näkymät ovat asiakkaalla, joka on tässä tapauksessa työssä toteutettu Angular-sovellus. Palvelun kontrollerit toimivat REST-rajapintoina, jotka tekevät tarvittavia toimenpiteitä, kuten esimerkiksi muilta mikropalveluilta datan hakemista tai datan manipuloimista. Kontrollerit palauttavat HTTP-vastauksissa datan mallien määrittelemässä muodossa. Näkymä ja kontrollerit ovat riippuvaisia mallien määrittelyistä, mutta mallit eivät ole riippuvaisia näkymistä tai kontrollereista. Tämä mahdollistaa niin kutsutun Separation of Concerns -suunnitteluperiaatteen, jossa ohjelma on jaettu

osiin, joista jokainen käsittelee erillistä ongelmaa tai tehtävää. Se tekee ohjelman kehittämisestä helpompaa, sillä jokaista eri osiota voidaan testata erillisinä kokonaisuuksina. Alla olevassa kuvassa 3 on esitetty hallintatyökalupalvelun puolella tapahtuva HTTP-pyynnön käsittelyn koko ketju.



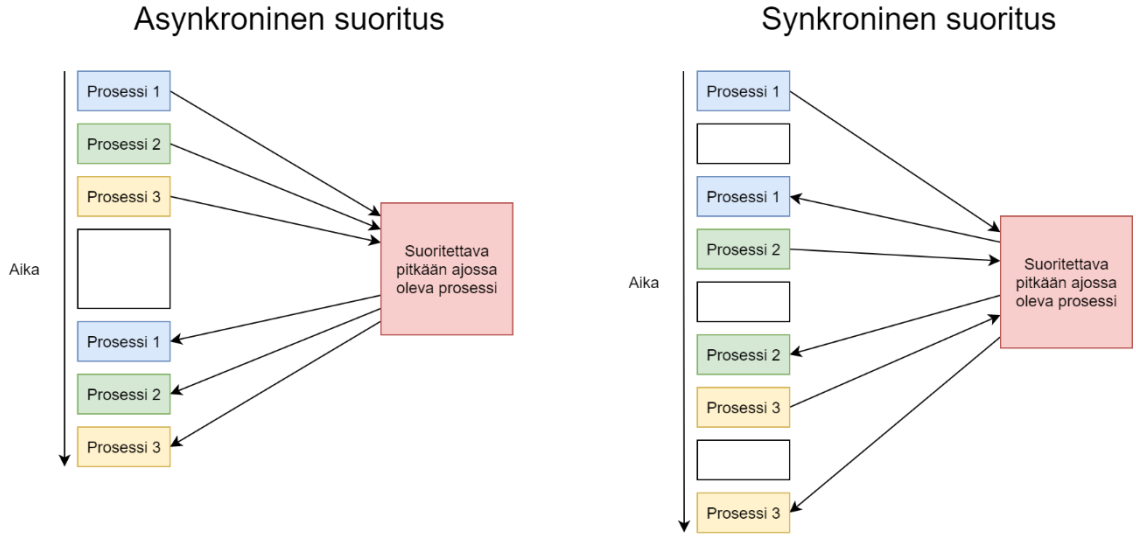

***Kuva 3.*** *Back-endin puolella tapahtuva hallintatyökalupalvelun työnkulku.*

Yllä oleva kuva 3 esittää työssä toteutetun Angular-sovelluksen tekemää HTTP-pyyntöä back-endiin. Esimerkkinä työnkulusta voidaan pitää käyttötapausta, jossa käyttäjä haluaa ottelun kokoonpanot tietylle ottelulle. Front-end lähettää HTTP-pyynnön back-endiin, jossa on mukana käyttäjän tarjoama ottelun yksilöivä *ottelu-ID* HTTP-parametrina. Ottelu ID:tä käytetään jokaisen ottelun yksilöimiseen. Hallintatyökalupalvelussa LeagueDataController toimii käsittelijänä pyynnölle, johon pyyntö ohjataan ASP.NET:n sisäänrakennettujen väliohjelmistokäsittelyjen jälkeen. Hallintatyökalussa ei käytetä räätälöityjä väliohjelmistoja, vaan siinä käytetään ASP.NET Core:n tarjoamia väliohjelmistoja, jotka suoritetaan ennen ja jälkeen kontrollerin funktion kutsua. Koska hallintatyökalupalvelulla ei ole omaa tietokantaa, se hakee halutun tiedon muista back-endin palveluista. Tässä esimerkkitapauksessa tarvittava tieto saadaan LeagueData-palvelulta. Hallintatyökalupalvelussa käytetään erillisiä kommunikaattoriluokkia, jotka abstrahoivat kaiken kommunikointilogiikan kontrollereilta. Niiden vastuulla on siis hakea dataa muista back-endin palveluista, mikäli hallintatyökalupalvelu tarvitsee dataa muualta. Kommunikaattoriluokissa luodaan gRPC-asiakas, eli olio, joka luo tyngät gRPC-palvelimella ajettavista RPC-palvelun funktioista. Kun tällaista tynkää kutsutaan asiakaspalvelussa, niin gRPC-kehys ajaa kohdepalvelussa vastaavan funktion toteutuksen. Kohdepalvelu palauttaa halutun resurssin, eli tässä tapauksessa ottelun kokoonpanot. Kommunikaattori-

luokka ei tee dataa manipuloivia operaatioita, vaan palauttaa datan sellaisenaan kontrollerille. Kontrolleri tekee tarvittavat jatkotoimenpiteet, joita ei tässä tapauksessa ole datan konversion lisäksi. Kontrolleri konvertoi datan ottelukokoonpano-mallin määrittelemään muotoon, ja palauttaa datan JSON-serialisoituna Angular-sovellukselle. Kontrolleriluokat ja kommunikaattoriluokat esitetään luvussa 5.2.

### 5.1.2 Asynkronisuus

Hallintatyökalun luokat on toteutettu siten, että lähes kaikki niiden funktiot ovat asynkronisia. Asynkronisuus tarkoittaa sitä, että ohjelmakoodi aloittaa jonkin pitkään ajossa olevan operaation, mutta ei jää odottamaan sen valmistumista. Synkronisessa suorituksessa ohjelmakoodi jää odottamaan prosessin valmistumista, ennen kuin seuraavan prosessin suoritus voidaan aloittaa. Asynkronisuus usein mielletään monisäikeisenä suorituksena, mutta asynkronisuudella ei ole itse asiassa mitään tekemistä säikeiden kanssa, vaikka hyvin yleisesti asynkronisuutta hyödynnetään usealla suoritussäikeellä. Asynkroninen suoritus monella säikeellä toteutetaan usein siten, että jokainen tehtävä suoritetaan omalla säikeellä synkronisesti taustalla. Alla olevassa kuvassa 4 on esitetty esimerkki synkronisesta ja asynkroninen suorituksesta, kun kolme eri tehtävää pyrkii suorittamaan pitkään ajossa olevan operaation.




***Kuva 4.*** *Asynkroninen ja synkroninen suoritus yhdellä suoritussäikeellä.*

Työssä käytettiin hyväksi .NET:n asynkronista ohjelmointimallia (Task asynchronous programming model, TAP). Se abstrahoi asynkronisen ohjelmakoodin kirjoittamisen, jolloin ohjelmoija voi kirjoittaa asynkronista koodia synkronisella syntaksilla. Tällöin asynkroniset funktiot merkitään *async*-avainsanalla, ja kutsut asynkronisiin funktioihin palaut-

tavat .NET:n Task-objektin, joka sitoo funktion palautusarvon Task-objektin asynkronisen operaation funktioihin. Synkronisessa kutsussa suorittava säie estäisi kaiken muun suorituksen, mutta asynkronisessa kutsussa säie on vapaana tekemään muuta työtä sillä välin, kun asynkroninen tehtävä on suorituksessa. *Await*-avainsanalla odotetaan, että tehtävä suoritetaan loppuun, jolloin ohjelmakoodin suoritus jatkuu siitä samaisesta kohdasta, jossa kyseinen await-avainsana sijaitsee.

### 5.1.3 Välimuistin hyödyntäminen

Palvelussa hyödynnettiin välimuistia tilanteissa, joissa oli tarvetta hakea harvoin muuttuvaa tietoa usein muista palveluista. Sen avulla vähennettiin merkittävästi RPC-kutsujen määrää muihin palveluihin. Redis-datavarasto oli valittu järjestelmän hajautetuksi välimuistiksi, sillä se sopii hyvin mikropalveluarkkitehtuuriin ja pilviympäristöihin, ja Redis oli valmiiksi konfiguroituna järjestelmässä. Hajautetussa välimuistissa data säilyy vaikka yksittäiset mikropalvelut kaatuvat tai käynnistyvät uusiksi, sekä lisäksi se ei vie resursseja järjestelmän palveluilta.

## 5.2 Palveluun toteutetut luokat

Tässä luvussa käydään läpi toteutetut kontrolleri-, kommunikaattori-, sekä malliluokat. Luvussa ei tarkastella apuluokkia.

### 5.2.1 Hallintatyökalupalvelun kontrollerit ja kommunikaattorit

Hallintatyökalussa toteutettiin seitsemän eri kontrolleriluokkaa, jotka ovat nimetty niiden käyttötarkoitusten mukaisesti: *TagsController*-luokka vastaa järjestelmän tagien CRUD-operaatioista (Create, Read, Update, Delete) TagControl-mikropalvelulta. *TagMappingController* puolestaan vastaa tagimäärittelyistä. *LeagueDataControllerin* vastuulla on hakea LeagueData-palvelulta yleistä älykiekko-otteluihin liittyvää tietoa, kuten esimerkiksi liigakausia, otteluita, joukkueita, pelaajia, ja aikatauluja. *RecordingController* vastaa ottelutallenteiden uudelleenaloittamisesta sekä tilastolaskennan aloittamisesta manuaalisesti. *StatusControllerin* vastuulla on hakea tieto pelikiekon livetilasta Status-mikropalvelulta, sekä lisäksi sen pitää pystyä tarjoamaan mahdollisuus ylikirjoittaa pelikiekon status manuaalisesti. *CacheController*-luokan vastuulla on tyhjentää järjestelmän hajautetusta välimuistista sinne tallennettua dataa.

Jokaista hallintatyökalun tarvitsemaa palvelua kohden on toteutettu yksi proxyluokka. Esimerkiksi LeagueData-palvelun RPC-kutsuja varten on luotu *LeagueDataCommunica-*

*tor*-luokka. Näiden luokkien funktiot palauttavat mikropalveluilta saadun vastauksen lähestulkoon manipuloimattomana, paitsi jos datasta tarvitaan vain jokin osa hallintatyökalussa. Mahdollinen datan suodatus kuitenkin tapahtuu kontrolleriluokissa nykyisessä toteutuksessa. Alla olevassa ohjelmassa 1 on esitetty eräs TagMappingController-luokan funktio.

```
public async Task<IActionResult> GetMatchTagMapping(int matchId)
{
	var tagMappingResponse =
		await _tagControlCommunicator.GetMatchTagMapping(matchId);

	if (tagMappingResponse.HasValue)
	{
		return Ok(tagMappingResponse.Value.ToManagementTagMapping());
	}

	var error = tagMappingResponse.Error;
	Logger.LogError($"Fetching tag mapping for match {matchId} failed:
		{error.Message}");
	return error.ToManagementErrorResponse();
}
```

***Ohjelma 1.*** *Kontrollerin funktio, joka pyrkii hakemaan tagimäärittelyä ottelu ID:lle.*

Ohjelmassa 1 on esimerkkinä TagMappingController-luokan funktio, jossa pyritään hakemaan ottelu ID:n perusteella tagimäärittelyä. Funktion toteutus on hyvin suoraviivainen, sillä tehtävänä tässä on ainoastaan hakea tietoa. TagControlCommunicator-luokan taakse on abstrahoitu kaikki gRPC-kutsuun liittyvät asiat, kuten gRPC-kanavan ja -asiakastyngän luonti, sekä itse gRPC-kutsu TagControl-palveluun. Vastauksena palautetaan HTTP-vastaus 200, mikäli tagimäärittelyn haku onnistui. Alla olevassa ohjelmassa 2 on esitetty vastaavan TagControlCommunicator-luokan yksinkertaistettua ohjelmakoodia.

```
public async Task<Faultable<MatchTagMapping>> GetMatchTagMapping(int matchId)
{
	_tagServiceClientFactory.CreateClient(
		out var tagServiceClient, out var tagServiceChannel);

	try
	{
		var tagMappingResponse = await tagServiceClient.GetTagMappingAsync(
			new GetTagMappingRequest { matchId = matchId });

		if (tagMappingResponse.Error != null)
		{
			return Faultable<MatchTagMapping>
				.WithError(tagMappingResponse.Error);
		}

		return Faultable<MatchTagMapping>
				.WithValue(tagMappingResponse.TagMapping);
	}
	catch (Exception ex)
	{
		return Faultable<MatchTagMapping>
			.WithError(ErrorResponses.InternalError());
	}
	finally
	{
		await tagServiceChannel.ShutdownAsync();
	}
}
```

***Ohjelma 2.*** *Kommunikaattorin funktio, joka hakee tagimäärittelyä gRPC:n yli.*

Ohjelmassa 2 haetaan ohjelman 1 kutsulle tagimäärittelyä RPC-kutsulla toisesta palvelusta. Siinä luodaan ensimmäisenä kanava sekä tynkä, jonka avulla kutsutaan TagControl-palvelun päässä olevaa toteutusta. Vastausta ei muuteta vielä tässä vaiheessa mallien määrittelemään muotoon, vaan se hoidetaan kontrolleritoteutuksen puolella ennen kuin palautetaan HTTP-vastaus käyttäjälle. Ohjelmat 1 ja 2 ovat yksinkertaistettuja esimerkkejä oikeasta toteutuksesta, josta on lisäksi karsittu esimerkiksi lokitukseen liittyvät logiikat pois.

Kaikkien kontrolleri-, sekä kommunikaattoriluokkien toteutukset ovat enimmäkseen hyvin suoraviivaisia. Ensin validoidaan käyttäjän antama syöte, jonka jälkeen haetaan tai syötetään dataa muihin mikropalveluihin kommunikaattoriluokkien avulla. Datan käsittelyn jälkeen käyttäjälle palautetaan HTTP-vastaus, joka on suurimmassa osassa käyttötapauksista konvertoitu hallintatyökalulle sopivaksi dataksi, eli mallien määrittelemiin muotoihin. Tällainen yksinkertainen käsiteltävä HTTP-pyyntö soveltuu tilanteisiin, joissa dataa tarvitaan vain keskitetystä back-endistä. Osa palveluista sijaitsee kuitenkin tietyn joukkueen back-endissa, jolloin toteutus ei ole näin yksinkertainen. Esimerkiksi ottelun tilastolaskenta suoritetaan työn kirjoitushetkellä pääsääntöisesti kotijoukkueen back-endissa. Tätä varten hallintatyökalupalvelu tarvitsee kyseisen back-endin osoitteen, jotta

tilastolaskenta voidaan aloittaa hallintatyökalusta käsin. Tällaista tilannetta varten järjestelmään on toteutettu BackendRouter-palvelu, joka sisältää tiedon jokaisen palvelun sisäisestä gRPC-osoitteesta ja -portista. Tämä tarkoittaa hallintatyökalupalvelun näkökulmasta sitä, että kutsuissa sen täytyy ensin hakea BackendRouter-palvelulta kyseinen osoite ja portti, ennen kuin kyseiseen palveluun voidaan tehdä gRPC-kutsu. Tässä käytettiin hyväksi välimuistia, sillä back-endin ja palvelun osoitteet eivät vaihdu jatkuvasti.

## 5.3 Asiakassovellus

Hallintatyökalun asiakassovellus toteutettiin *Single-Page Application* (SPA) -sovelluksena. SPA-sovelluksissa sivusto sisältöineen ladataan vain kerran, jolloin asiakassovellus saa koko sivuston sisällön käyttöönsä välittömästi. Tällöin asiakassovelluksen tarvitsee huolehtia ainoastaan sisällön esittämisestä käyttäjälle. Sisällöllä tässä tarkoitetaan HTML-, JavaScript-, tyylitiedostoja, sekä muita mahdollisia staattisia tiedostoja, kuten esimerkiksi kuvatiedostoja. Asiakkaan ja palvelimen välillä kulkee ainoastaan dataa, jonka tarkoituksena on täydentää asiakassovelluksen esittämä sisältö. Tällöin datansiirto on suhteellisen vähäistä, sillä sisältöä ei tarvitse joka kutsulla hakea ja ladata uudelleen asiakaspäässä. Hallintatyökalu toteutettiin luvussa 4 esitetyllä Angular-sovelluskehyksellä. Työssä käytettiin työn tekohetkellä Angularin versiota 7.

Hallintatyökalusovelluksessa käytettiin hyväksi RxJS-kirjastoa [25], joka helpottaa asynkronisen ja tapahtumapohjaisen ohjelmakoodin muodostamista web-sovelluksissa. Sen pääasiallinen tarkoitus on siis käsitellä asynkronisia tapahtumia, jotka käytännössä tuottavat datavirtaa. RxJS:ssa asynkronista datavirtaa esitetään tarkkailijoiden avulla (observer). Tarkkailija toimii siten, että se rekisteröityy kuuntelemaan datavirran lähdettä (subscribe), ja se reagoi siihen heti, kun uutta dataa lähetetään. Sen avulla työkalussa voitiin siis tehdä useampi HTTP-kutsu ketjuttamalla ne RxJS:n tarjoamilla tyypeillä ketjuksi. Alla olevassa ohjelmissa 3 ja 4 on esitetty hallintatyökalun front-endin tilanne, jossa haetaan back-endilta tagimäärittelyä.

```
public getMatchTagMapping$( matchId: number ): Observable<TagMapping[]> {
    const url = `${this.tagMappingUrl}/match/${matchId}`;
    return this._httpClient.get<TagMapping[]>(url);
}
```

***Ohjelma 3.*** *HTTP-kutsun suorittava funktio, joka palauttaa sen uutena Observable-tietotyyppinä.*

Ohjelma 3 tekee HTTP-kutsun ohjelmassa 1 esitettyyn kontrolleriin. Tässä palautetaan back-endilta saatu HTTP-vastaus Observable-tyyppiin sidottuna. Tyypillisesti dollarimerkillä osoitetaan sitä, että kyseinen funktio palauttaa Observable-tietotyyppiä paluuarvonaan.

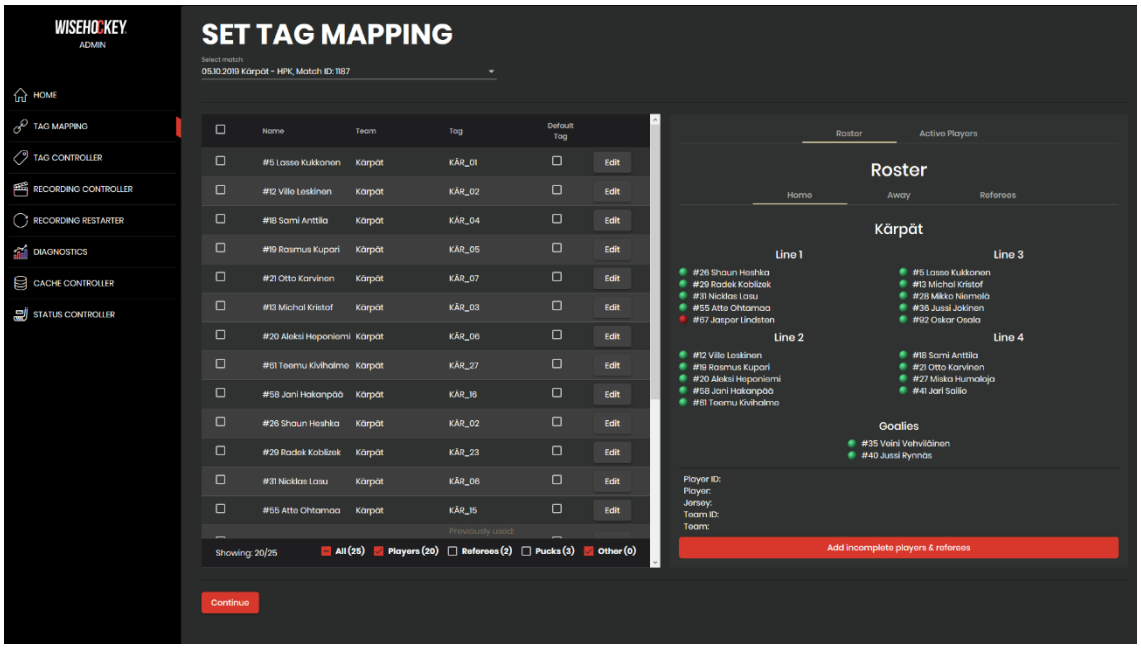
```
public getMatchTagMappings() {

    ...

    // Fetch tag mapping
    this._tagCommunicatorService.getMatchTagMapping$( this.selectedMatch.id
).pipe(
        mergeMap( ( tagMapping: TagMapping[] ) => {
            ...
            this._tagMapping = tagMapping;
        } ),
        catchError( error => {
            ...
        } )
    )
}
```

***Ohjelma 4.*** *Kommunikaattorin funktio, joka hakee tagimäärittelyä gRPC:n yli.*

Ohjelmassa 4 esitetään esimerkkitilanne, jossa haetaan back-endilta tagimäärittelyä. Observable-tietotyyppi mahdollistaa sen, että käyttöliittymä voidaan täydentää samantien *callback*-funktion avulla, joka tässä tapauksessa on *mergeMap*-operaattorin sisällä määritelty anonyymi *nuolifunktio*. Esimerkissä vain asetetaan kyseisen luokan muuttujan arvoksi vastauksen paluuarvo, joka Angularin kahdensuuntaisella sidonnalla saadaan vietyä käyttöliittymään automaattisesti.

### 5.3.1 Ottelun tagihallinta

Työssä toteutettu ottelun tagihallintanäkymä on hallintatyökalun oleellisin työkalu ottelupäivinä. Sen pääasiallinen käyttötarkoitus on tarjota mahdollisuus korjata otteluiden puutteellisia tagimäärittelyjä käsin. Tagimäärittely on puutteellinen, mikäli jollekin kokoonpanon pelaajista tai ottelun kiekoista ei ole määritetty tagia. Tagimäärittely tehdään myös tuomareille, mutta heille määrittely ei ole tällä hetkellä pakollista. Tagimäärittelytyökalun näkymästä on tehty mahdollisimman yksinkertainen, jotta päivystäjien tekemät virheet olisivat mahdollisimman minimaalisia. Alla olevassa kuvassa 5 on esitetty hallintatyökaluun toteutettu tagimäärittelynäkymä.




***Kuva 5.*** *Tagimäärittely-työkalun päänäkymä.*

Kun käyttäjä navigoi sovelluksessa tagimäärittelynäkymään, niin sovellus hakee back-endin hallintatyökalupalvelulta HTTP-kutsulla päivystäjän kannalta kiinnostavat ottelut. Käytännössä nämä ottelut ovat kyseisen päivän otteluita, jotka pelataan älykiekkoa tukevissa halleissa, ja ottelun kotijoukkueena toimii jokin älykiekkoa käyttävistä joukkueista. Ottelut voivat myös olla käynnissä olevia otteluita, sillä ottelun aikanakin voi tapahtua tagimuutoksia, esimerkiksi jos pelaajalle määritetty tagi hajoaa. Kokoonpanomuutokset eivät kuitenkaan ole mahdollisia ottelun alettua. Pudotusvalikossa esitetään haetut ottelut, joista käyttäjä valitsee ottelun, johon hän haluaa tehdä muutoksia. Sovellus hakee tämän lisäksi myös kiinnostavien otteluiden kokoonpanot tässä vaiheessa. Ottelukokoonpanoon on sisällytetty mukaan myös tuomarit. Kiekot eivät ole mukana ottelukokoonpanoissa, sillä kiekkojen yhdistäminen otteluun on triviaalia. TagControl-palvelulla on jo tieto ottelun kiekoista, sillä kiekot ovat aina hallikohtaisia.

Kun käyttäjä on valinnut ottelun, niin sovellus hakee back-endilta kyseisen ottelun tagimäärittelyn. Sovellus tekee siis HTTP-kutsun hallintatyökalupalveluun, joka tekee RPC-kutsun TagControl-palveluun. Tagimäärittelyn haettuaan sovelluksessa vertaillaan tagimäärittelyä ottelun kokoonpanoon, jonka perusteella päätellään puutteelliset pelaajat ja tuomarit. Sovellus hakee puutteellisille pelaajille hallintatyökalupalvelulta tiedon heidän viimeksi käyttämistään tageistaan kyseisessä joukkueessa, koska hyvin usein heidän viimeksi käyttämät tagit ovat samoja tageja, joilla heidän on tarkoitus pelata.

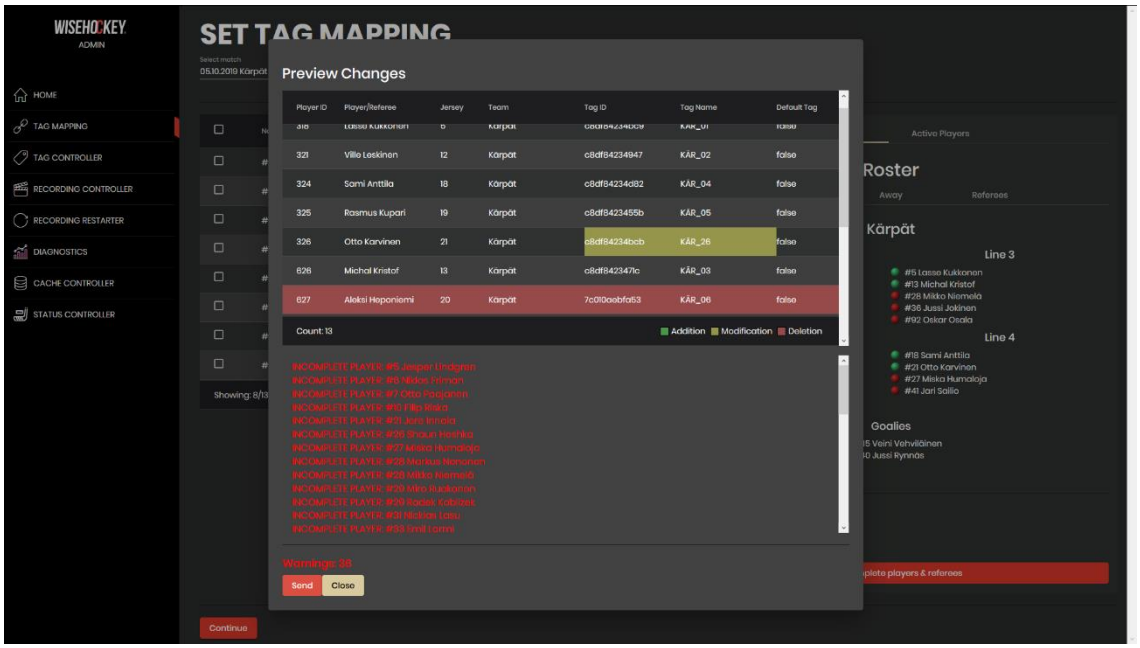
Sovellus täyttää kuvan 5 vasemmalla puolella olevan taulukon haetulla tagimäärittelyllä. Näkymän oikealla puolella on näkyvillä valitun ottelun kokoonpanot. Kokoonpanoissa olevien pelaajien vasemmalla puolella oleva indikaattori kertoo alustavasti, onko pelaaja

määritetty onnistuneesti. Tarkoituksena on, että kaikki kokoonpanossa olevat puutteelliset pelaajat (ja tuomarit) lisätään vasempaan taulukkoon, jonka jälkeen heille asetetaan tagit. Mikäli kokoonpanon määrittelemätön pelaaja on joskus pelannut kyseisessä joukkueessa, niin taulukossa hänelle ehdotetaan viimeisintä tagia, millä hän on pelannut kyseisessä joukkueessa. Ehdotusta ei tehdä, mikäli kyseinen tagi on jollakin nykyisellä pelaajalla käytössä. Tagien asetusta varten sovellus hakee hallintatyökalupalvelulta listan kaikista järjestelmän tageista. Tässä haetaan kaikki järjestelmän tagit siksi, että vaatimuksen mukaan pelaajille ja tuomareille pitää pystyä työkalun avulla asettamaan mikä tahansa järjestelmän tagi. Käytännössä tuomareille ei ole mahdollista asettaa kiekon tagia, mutta tämä tilanne tarkastetaan back-endissa.

Tagien asetusten jälkeen käyttäjälle näytetään modaalinäkymä, jossa varmistetaan käyttäjän tekemät muutokset. Päivystäjä pystyy tarkistamaan näkymästä, mitä muutoksia hän on kullekin pelaajalle, tuomarille, ja mahdollisesti kiekolle tehnyt. Näkymässä näytetään samainen taulukko, mutta muutokset näytetään eri väreillä:

- Vihreällä värillä indikoidaan uutta lisäystä tagimäärittelyyn, eli siihen on lisätty uusi pelaaja, tuomari, tai kiekko.
- Punaisella värillä indikoidaan poistoa tagimäärittelystä.
- Keltaisella värillä indikoidaan yksittäistä muutosta. Esimerkiksi jos pelaajan tagi on vaihdettu, niin kyseisen pelaajan taulukon rivin tagisarakkeiden solut ovat keltaisella pohjalla.

Taulukon lisäksi näkymässä näytetään listana kaikki käyttäjän tekemät muutokset selkotekstinä, sekä mahdolliset puutteet, mitä käyttäjän tekemissä muutoksessa on. Puute voi olla esimerkiksi tapaus, jossa käyttäjä ei ole määritellyt pelaajalle tagia ollenkaan.




***Kuva 6.*** *Tarkistusnäkymä virheiden ehkäisemiseksi.*

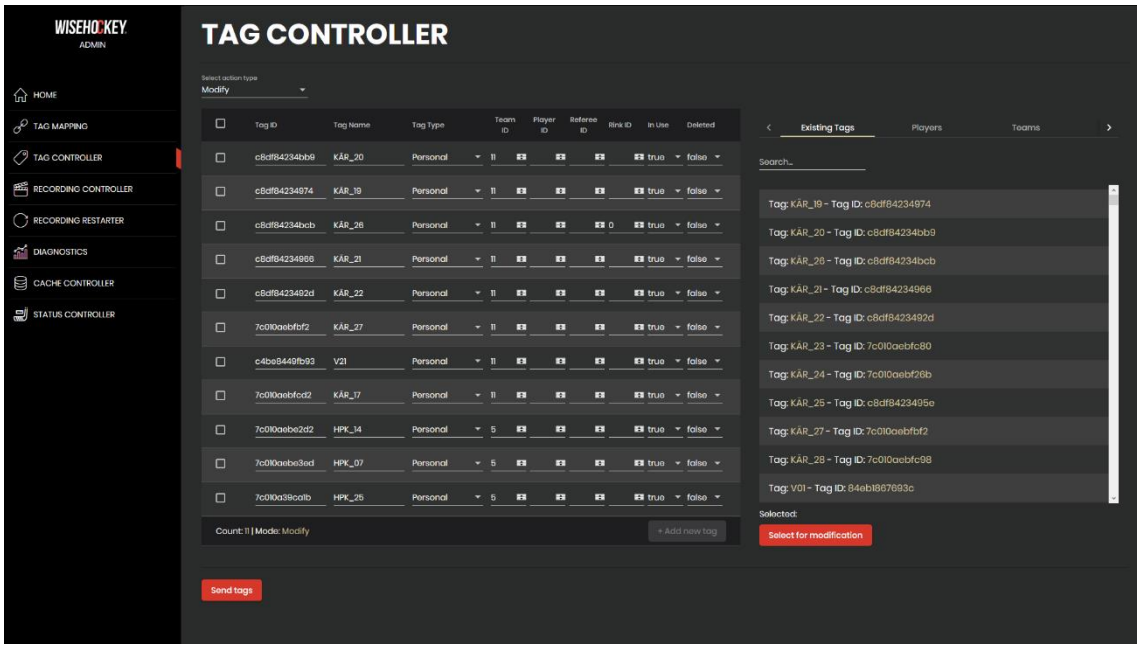
Kuvassa 6 on esitetty tagimäärittelyn tarkastusnäkymä, jonka tarkoituksena on estää, ettei käyttäjä vahingossa lähetä keskeneräistä tagimäärittelyä eteenpäin.

### 5.3.2 Tagikontrollointi

Tagien kontrollointinäkymän tarkoituksena on toimia tagien hallintanäkymänä, eli sen avulla pitää pystyä suorittamaan luku- ja kirjoitusoperaatioita tageille. Tämän hetkinen toteutus mahdollistaa sen, että tageja voidaan lisätä järjestelmään, niihin liittyvää tietoa voidaan muuttaa järjestelmässä, sekä tageja voidaan asettaa poistetuiksi järjestelmässä. Tagien poisto ei tässä tapauksessa tarkoita tagien poistamista järjestelmän tietokannasta, vaan se tarkoittaa, että tagi asetetaan piilotetuksi järjestelmän muilta osilta. Tästä käytetään nimitystä *pehmeä poisto* (soft delete). Tagi itsessään koostuu ainoastaan yksilöivästä ID:sta sekä nimestä, mutta tageihin liittyy muutakin järjestelmän toteutuksen kannalta kiinnostavaa tietoa;

- Hallin ID: tarkoitettu kiekkotageille, sillä kiekot ovat hallikohtaisia.
- Pelaaja/tuomari ID: tarkoitettu pelaajille sekä tuomareille. Molemmat ovat tietokannassa omina kenttinään, joista vain toisessa on jokin validi arvo.
- Tagin tyyppi: esittää joko henkilökohtaista tagia tai kiekkotagia.
- Joukkueen ID: tarkoitettu pelaajille. Tämä tieto on olemassa siksi, että henkilökohtaiset tagit jaetaan joukkueille, jotka jakavat tagit reaalimaailmassa ne pelaajilleen.

- Tieto siitä onko tagi aktiivisesti käytössä.
- Tieto siitä onko tagi poistettu käytöstä.



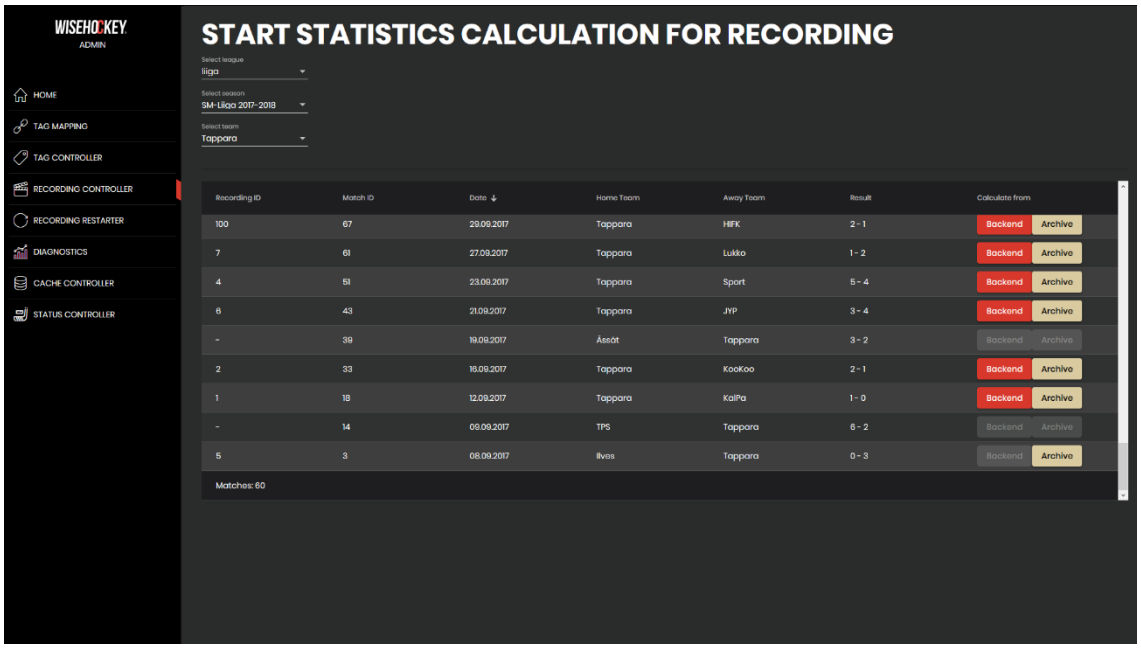

***Kuva 7.*** *Tagien kontrollointinäkymä.*

Yllä olevassa kuvassa 7 on esitetty itse tagien kontrollointinäkymä. Näkymän vasemmalla puolella on samantyylinen taulukko kuin tagimäärittely-näkymässä, mutta tässä taulukossa sen sarakkeet liittyvät tagitietoihin. Kun käyttäjä navigoi näkymään, sovellus hakee kaikki käytössä olevat tagit back-endistä, ja ne esitetään näkymän oikealla puolella olevassa HTML-elementissä. Käyttäjä voi lisätä näitä tageja vasempaan taulukkoon muutoksia varten klikkaamalla tagia. Sovellus myös hakee kaikkien aktiivisena olevien liigojen pelaajatiedot, tuomaritiedot, sekä joukkueet. Nämä tiedot auttavat käyttöliittymässä tagien hallinnassa, sillä tietokannassa tagiin liittyvät tiimi, halli, pelaaja tai tuomari ovat ID:na, jotka eivät sellaisenaan hyödytä käyttäjää käyttöliittymässä. Pudotusvalikosta käyttäjä voi valita halutun operaation, joka tullaan suorittamaan tageille. Operaatioita on kolme; tagien lisäys järjestelmään, tagien muuttaminen, ja tagien pehmeä poisto.

Tagien kontrollointinäkymän taulukko voidaan myös täyttää lukemalla tagit Quupan määrittelemästä JSON-tiedostosta. Tämä on hyödyllistä, mikäli käyttäjä haluaa lisätä kerralla suuren määrän tageja järjestelmään. Tämä JSON-tiedosto pitää sisällään tiedon tagin tyypistä, nimestä, ID:sta, sekä muutamasta muusta tiedosta, jotka eivät ole mielenkiintoisia hallintatyökalun näkökulmasta.

### 5.3.3 Älykiekkotilastojen uudelleenlaskenta

Älykiekkotilastojen uudelleenlaskenta on operaatio, joka joudutaan tekemään, mikäli älykiekkotilastojen laskentaan sisältyviin algoritmeihin tehdään korjauksia tai muita muutoksia, jotka vaikuttavat tilastolaskennan tuloksiin. Tätä varten hallintatyökaluun haluttiin näkymä laskennan uudelleenaloitukseen. Alla olevassa kuvassa 8 on esitetty uudelleenlaskentanäkymä.



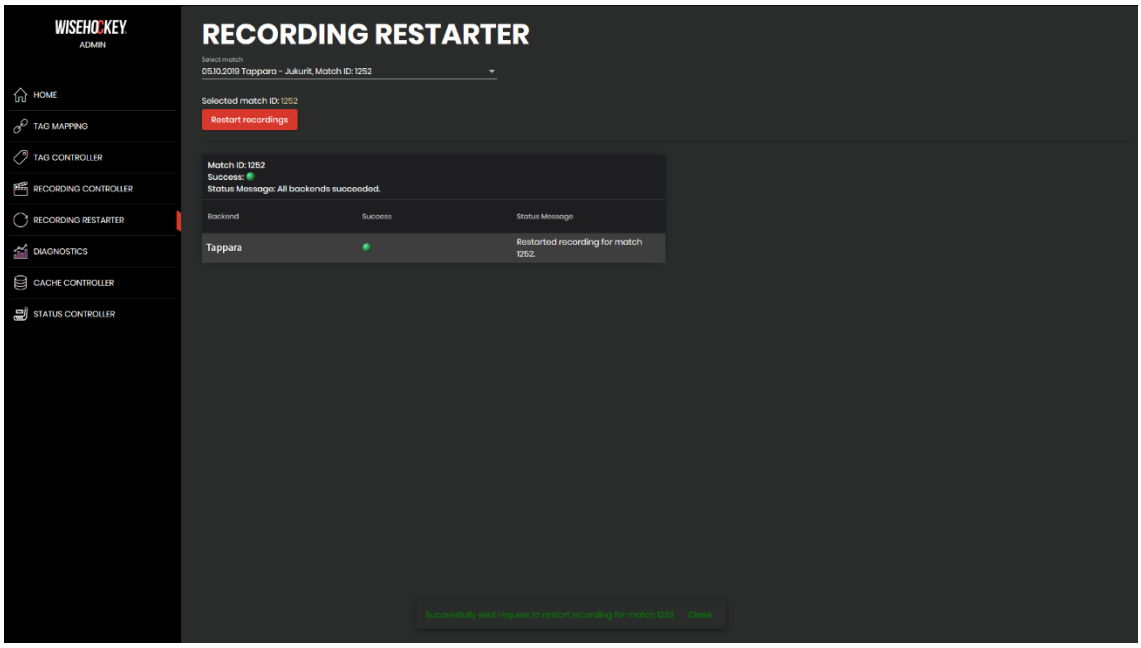

***Kuva 8.*** *Älykiekkotilastojen uudelleenlaskennan päänäkymä.*

Tilastolaskenta aloitetaan uudelleen aina joukkueen back-endissa, joten näkymässä tarvitaan ensimmäiseksi tieto joukkueista, joille voidaan tilastolaskenta ylipäänsä suorittaa. Näkymän yläreunassa on pudotusvalikot, joista valitaan haluttu liiga, kausi ja joukkue. Näkymässä on taulukko, jonne haetaan back-endilta kaikki valitun kauden ottelut, joissa valittu joukkue on pelannut. Viimeisessä sarakkeessa on jokaisen ottelun kohdalla kaksi painiketta, joista valitaan raakadatan lähde. Ottelun aikana kerätty raakadata tallennetaan sekä joukkueen back-endiin, että erilliseen data-arkistoon. Hallintatyökalulla voidaan siis aloittaa tilastolaskenta uudelleen käyttäen dataa data-arkistosta, tai vastaavasti joukkueen back-endista, jolloin joukkueen back-end saa datansa Kafkasta.

### 5.3.4 Tallenteen uudelleenaloitus

Järjestelmän back-endissa on tilanteita, jolloin aktiivinen tallenne terminoidaan, eikä tallennetta aloiteta automaattisesti uudelleen. Tällaista tilannetta varten hallintatyökaluun toteutettiin yksinkertainen näkymä tallenteen uudelleenaloitusta varten. Aktiivinen tallenne tarkoittaa sitä, että järjestelmä pyrkii keräämään raakadataa ja laskemaan sille älykiekkotilastoja. Yhtä ottelua kohden voi olla monta aktiivista tallennetta, mutta vain yksi aktiivinen tallenne kerrallaan yhtä back-endia kohden. Työn kirjoitushetkellä on

mahdollista, että ottelun koti- ja vierasjoukkueella molemmilla on olemassa älykiekkoa tukeva back-end, jolloin molemmilla back-endeilla on aktiivinen tallenne samasta ottelusta käynnissä. Näkymässä näytetään tallenteen uudelleenaloituksen tulos, eli onnistuiko tallenteen uudelleenaloitus back-endeissa. On mahdollista, että vain toisen back-endin tallenteen uudelleenaloitus onnistuu, jolloin operaatio täytyy tehdä uudelleen.

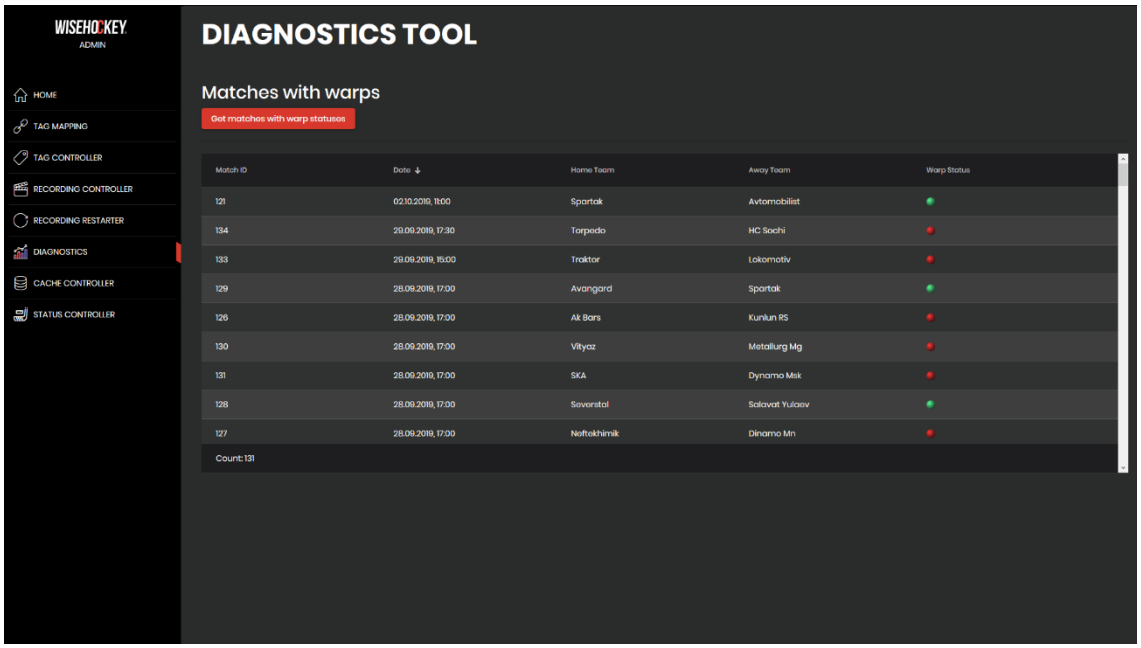



***Kuva 9.*** *Tallenteen uudelleenaloituksen päänäkymä.*

Yllä olevassa kuvassa 9 on esitetty tallenteen uudelleenaloituksen päänäkymä. Se on hyvin yksinkertainen, sillä suurin työ tehdään back-endissa. Näkymässä valitaan haluttu ottelu, jolloin tallenne voidaan aloittaa uudelleen valitulle ottelulle. Näkymässä näytetään myös tallenteen uudelleenaloittamisen tulos jokaiselle back-endille. Näin tehdään siksi, että voidaan helposti katsoa, mikäli jokin tallenteen uudelleenaloitus epäonnistuu jossakin tietyssä back-endissä.

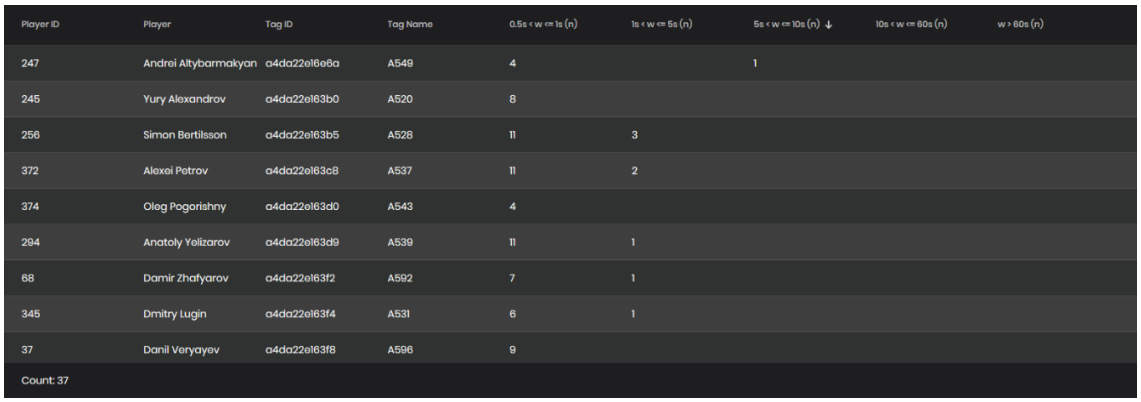
### 5.3.5 Tagien datapisteväärистymät

Hallintatyökalulla haluttiin visualisoida tagien vääristymiä yksittäisten raakadatapisteiden välillä. Tagin vääristymä tarkoittaa siis kahden peräkkäisen lähetetyn signaalin, eli datapisteen, välistä aikaerotusta, jotka vaihtelevat pienestä latenssista ja datapisteitä vastaanottavien lokaattoreiden sijainnista ja suuntauksesta johtuen. Työkalun tarkoituksena on kuvata tagien toimintavarmuutta, sillä tagien toimintaa voidaan arvioida aikaleimojen erotuksen avulla; tageista saatava data on vääristynyttä, mikäli niistä ei saada paikkadataa riittävällä intervallilla. Esimerkiksi jos tagin kahden peräkkäisen datapisteen väliset aikaleimat ovat useita sekunteja erillä toisistaan, niin todennäköisesti tagissa on vikaa. Alla olevassa kuvassa 10 on esitetty niin sanottu diagnostiikkanäkymä.




***Kuva 10.*** *Diagnostiikkanäkymän ottelulistaus.*

Näkymässä on taulukko, joka täytetään back-endilta saadun vastauksen perusteella. Taulukossa näytetään kaikki ottelut, joissa on vääristymiä olemassa. Käyttäjä voi avata ottelun vääristymien tiedot painamalla taulukon riviä, jolloin sovellus hakee back-endilta kyseisen ottelun vääristymät. Kuten kuvassa 11 esitetään, nämä vääristymät muunnetaan asiakaspäässä sellaiseen muotoon, että käyttöliittymässä näytetään jokaisen tagin vääristymien lukumäärät tiettyjen aikavälirajojen sisällä, kuten esimerkiksi vääristymien lukumäärät aikavälillä 0.5s-1.0s. Nämä rajat ovat määritelty back-endissa, ja ne ovat työn kirjoitushetkellä 0.5s, 1.0s, 2.0s, 5.0s, 10.0s, yli 60.0s. Mikäli halutaan tarkastella yksittäisen tagin kaikkia vääristymiä, eikä vain rajojen sisällä olevia lukumääriä, niin käyttäjä voi halutessaan avata kaikki vääristymät painamalla jotakin riviä taulukosta, jolloin näytetään kyseisen rivin tagin kaikki ottelun vääristymät kuvan 12 mukaisesti. Yksittäinen vääristymä käsittää vääristymän aloituskoordinaatin, loppukoordinaatin, alkuajan, sekä loppuajan. Yli kahden sekunnin vääristymät ovat käyttöliittymässä korostettu punaisella värillä niiden erottamiseksi.



| Player ID | Player | Tag ID | Tag Name | 0.5s < w <= 1s (n) | 1s < w <= 5s (n) | 5s < w <= 10s (n) ↓ | 10s < w <= 60s (n) | w > 60s (n) |
|---|---|---|---|---|---|---|---|---|
| 247 | Andrei Altybarmakyan | a4da22e16e6a | A549 | 4 | | 1 | | |
| 245 | Yury Alexandrov | a4da22e163b0 | A520 | 8 | | | | |
| 256 | Simon Bertilsson | a4da22e163b5 | A528 | 11 | 3 | | | |
| 372 | Alexei Petrov | a4da22e163c8 | A537 | 11 | 2 | | | |
| 374 | Oleg Pogorishny | a4da22e163d0 | A543 | 4 | | | | |
| 294 | Anatoly Yelizarov | a4da22e163d9 | A539 | 11 | 1 | | | |
| 68 | Damir Zhafyarov | a4da22e163f2 | A592 | 7 | 1 | | | |
| 345 | Dmitry Lugin | a4da22e163f4 | A531 | 6 | 1 | | | |
| 37 | Danil Veryayev | a4da22e163f8 | A596 | 9 | | | | |

Count: 37

***Kuva 11.*** *Ottelun vääristymien määrät.*

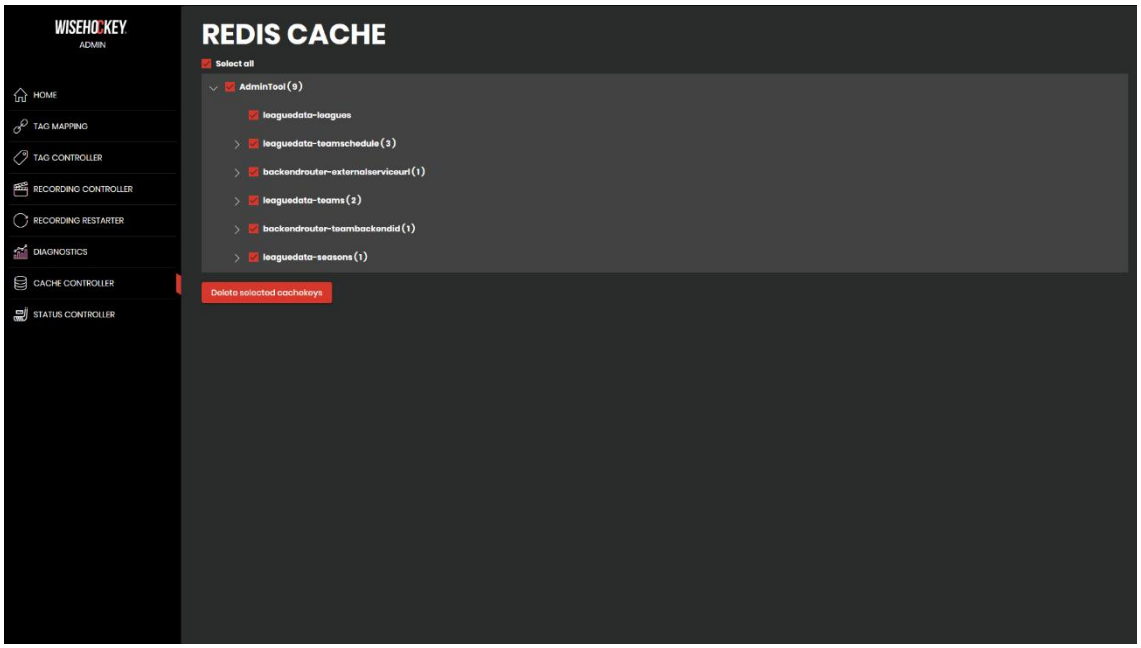
| Start TS (ms) | End TS (ms) | Delta TS (ms) ↓ | Start X coord. | Start Y coord. | End X coord. | End Y coord | Type |
|---|---|---|---|---|---|---|---|
| 1569772590351 | 1569772598733 | 8382 | -29.93 | -5.99 | -0.95 | -0.29 | OverFiveSeconds |
| 1569775509992 | 1569775510761 | 769 | -19.01 | -12.81 | -23.74 | -8.86 | OverHalfASecond |
| 1569775285987 | 1569775286521 | 534 | 2.8 | -10.38 | -0.42 | -8.09 | OverHalfASecond |
| 1569771975009 | 1569771975535 | 526 | 3.34 | 14.53 | 4.75 | 13.4 | OverHalfASecond |
| 1569772587627 | 1569772588134 | 507 | -19.24 | -13.6 | -21.7 | -14.06 | OverHalfASecond |

Count: 5

***Kuva 12.*** *Yksittäisen tagin kaikki huomattavat vääristymät.*

### 5.3.6 Välimuistin tyhjennys

Varsinkin järjestelmän kehityksessä sekä pilviympäristöä testattaessa välimuistissa tallennettuja avaimia halutaan voida poistaa tarpeen mukaan. Tätä varten hallintatyökaluun toteutettiin käyttöliittymä, jonka avulla välimuistin avaimien poistaminen on mahdollista. Tämä toiminto toteutettiin lisäominaisuutena, sillä tämä ominaisuus ei tule päivystäjien käyttöön, vaan ainoastaan järjestelmän kehittäjille. Alla olevassa kuvassa 13 on esitetty välimuistin tyhjennystä varten toteutettu näkymä.




***Kuva 23.*** *Välimuistin tyhjennys onnistuu tämän näkymän avulla.*

Välimuistin avaimet tallennetaan välimuistiin siten, että avain sisältää suorittavan palvelun nimen, jonka perään liitetään yksi tai useampi resurssi kaksoispisteellä eroteltuna. Back-endissa avaimet haetaan välimuistista, joista muodostetaan puumainen rakenne käyttöliittymää varten. Käyttöliittymässä avaimet esitetään puurakenteena käyttäen hyväksi Angular Material -kirjaston mat-tree-direktiiviä, jonka avulla esitetään hierarkkista dataa pudotusvalikoiden avulla. Näistä avaimista käyttäjä valitsee haluamansa avaimet, jotka halutaan poistaa. Kun poistettavat avaimet lähetetään takaisin back-endille, niin avaimet rakennetaan takaisin alkuperäiseen muotoonsa.

## 5.4 Testaus

Hallintatyökalun testausta varten luotiin uusi testipalvelu yksikkötestejä varten. Yksikkötestaus on ohjelmistotestausmenetelmä, jossa tyypillisesti kehittäjät kirjoittavat automatisoituja testejä, joiden tarkoituksena on varmistaa jonkin yksittäisen yksikön, kuten luokan tai funktion oikeanlainen käyttäytyminen [26]. Hyvin kirjoitetut yksikkötestit nostavat ohjelmiston laatua, sillä ne ennaltaehkäisevät kehittäjän tekemiä virheitä, ennen kuin ohjelmakoodi viedään tuotantoon asti. Yksikkötestit myös pakottavat kehittäjän miettimään testattavan yksikön toimintaa tarkemmin.

Hallintatyökalussa käytettiin hyväksi Moq-testauskehystä [27], joka on .NET-alustalle kehitetty testauskirjasto. Se on ainoa kirjasto .NET-alustalla, joka tukee kokonaisuudessaan *Language-Integrated Query* -lausekkeita (Linq) [28]. Hallintatyökalun testiprojektissa toteutettiin yksikkötestit back-endin kontrollereita varten. Nämä testit testaavat sen,

että kontrollerit palauttavat oikeaoppisen HTTP-vastauksen kunnollisella statuskoodilla, sekä halutuilla resursseilla. Alla olevassa ohjelmassa 3 on esitetty esimerkki yksikkötestistä.

```
public async void GetMatchTagMapping_WithBadMatchId_ShouldReturn500()
{
	// Setup
	var badMatchId = -1;
	_tagControlCommunicator.Setup(tcc =>
			tcc.GetMatchTagMappingAsync(badMatchId)
		.ReturnsAsync(Option.None());

	// Act
	var actionResult = await _controller.GetMatchTagMapping(badMatchId);

	// Assert
	Assert.NotNull(actionResult);
	Assert.IsType<BadRequestObjectResult>(actionResult);
	var result = Assert
		.IsAssignableFrom<BadRequestObjectResult>(actionResult);
	Assert.Equal(500, result.StatusCode);
}
```

***Ohjelma 5.*** *Yksikkötesti, jossa testataan tyhjää parametrilistaa.*

Ohjelmassa 3 on esitetty *GetMatchTagMapping*-funktion eräs yksikkötesti. Siinä testataan tilannetta, jossa kontrollerilta haetaan tagimäärittelyä ottelulle, jonka ID on -1. Testissä käytetään *_tagControlCommunicator*-valemuuttujaa, jolle määritetään paluuarvo, jonka se esittää palauttavansa, kun kontrollerin testi suoritetaan. Siinä ei siis oikeasti kutsuta *GetMatchTagMappingAsync*-funktiota, vaan valemuuttuja matkii oikean muuttujan toimintaa. Testattava funktio suoritetaan, kun tarvittavat järjestelyt on alustettu. Suorituksen jälkeen funktion paluuarvoa verrataan odotettuihin arvoihin ja varmistetaan, että palautusarvo ja odotetut arvot vastaavat toisiaan.

# 6. ARVIOINTI JA JATKOKEHITYS

## 6.1 Vaatimusten täyttyminen

Ensisijaisesti tavoitteena oli kehittää ottelun päivystäjän toimintaa tehostava työkalu, jonka avulla päivystäjät kykenisivät operoimaan järjestelmää tehokkaasti Liigan ottelupäivänä. Aluksi työkalun käyttökohderyhmä rajoittui ainoastaan älykiekkoprojektin jäseniin, sillä päivystäjät olivat tässä vaiheessa samalla järjestelmän kehittäjiä. Tämän takia toiminnallisiin vaatimuksiin tuli mukaan myös sellaisia ominaisuuksia, jotka eivät tulevaisuudessa saisi olla mahdollisille projektin ulkoisille sidosryhmille näkyviä. Esimerkiksi työkalulla kehittäjän piti pystyä vaikuttamaan järjestelmän eri tiloihin, kuten esimerkiksi tallenteiden ja tilastolaskennan uudelleenaloitukseen, sekä järjestelmän välimuistiin. Sen ainoa määritelty ei-toiminnallinen tavoite oli, että sen piti olla web-pohjainen. Lisäksi osa ei-toiminnallisista tavoitteista oli enemmän tai vähemmän määrittelemättömiä.

Aluksi ei ollut edes suunnitteilla, että työkalu tulisi projektin ulkoisille sidosryhmille käytettäväksi, mutta vuoden 2018-2019 vaihteessa työkalun tavoitteeksi asetettiin, että liigat pystyisivät operoimaan sillä itse kaudesta 2019-2020 eteenpäin. Tämä asetti työkalulle toiminnallisten vaatimusten lisäksi myös selkeän ei-toiminnallisen käytettävyysvaatimuksen. Sen täytyi olla niin helppokäyttöinen, että kuka tahansa järjestelmän toiminnasta tietämätön henkilö kykenisi suorittamaan työkalulta vaaditut toimenpiteet yleisen päivystyskoulutuksen jälkeen. Lisäksi sen käytettävyys piti olla sillä tasolla, että mahdolliset työkalun käytöstä koituvat virhetilanteet johtuisivat käyttäjän omasta huolimattomuudesta, eikä huonosta käytettävyydestä, työkalun suunnittelusta tai puutteellisesta järjestelmän toiminnasta.

Toiminnallisesta näkökulmasta tavoitteet saavutettiin. Työkalulla voidaan tehdä kaikki ne asiat, mitä kauden aikana sen toiminnallisiksi vaatimuksiksi asetettiin. Sillä voidaan asettaa manuaalisesti tagimäärittelyjä hyvinkin yksinkertaisesti. Yksinkertaisimmassa tapauksessa käyttäjän tarvitsee ainoastaan lisätä kokoonpanon pelaajat tagimäärittelyyn mukaan ja samalla asettaa heille oikeat tagit. Tagimäärittely saadaan tallennettua järjestelmään yksinkertaisella napin painalluksella. Sillä voidaan omassa näkymässään lisätä, poistaa ja muuttaa tageja. Sen avulla voidaan aloittaa käynnissä oleva tallenne uudelleen ja tilastolaskenta sekä Kafkaan tallennetusta datasta että data-arkiston datasta. Sillä voidaan myös tarkastella ottelun tagien vääristymiä sekä lukumäärällisesti aikarajojen puitteissa että yksityiskohtaisesti. Välimuisti voidaan tyhjentää myöskin omassa näkymässään yksinkertaisesti.

Hallintatyökalun toteutus oli alusta alkaen hyvin joustava ja iteratiivinen prosessi. Työkalun toiminnalliset vaatimukset muuttuivat jatkuvasti sitä mukaa, kun älykiekkojärjestelmään tuotiin uusia ominaisuuksia sekä olemassa oleviin ominaisuuksiin tehtiin muutoksia, jotka vaikuttivat työkaluun. Esimerkiksi prosessin alussa työkalun ainoa päivystystä helpottava toiminnallinen vaatimus oli, että kehittäjä voisi suorittaa web-käyttöliittymän kautta ottelun tagimäärittelyn, joka vielä prosessin alussa käsitti ainoastaan ottelun pelaajat sekä kiekot. Tuomarit eivät vielä siinä vaiheessa olleet tuettuina back-endissa ollenkaan. Tämän lisäksi monen muunkin uuden vaatimuksen kohdalla järjestelmän monista palveluista puuttui tuki kyseisen vaatimuksen toteuttamiselle, joten näissä tapauksissa piti myös tehdä toteutukset muihinkin palveluihin kuin hallintatyökaluun. Se kasvatti työmäärää suhteellisen paljon, joka hidasti työkalun toteutusta merkittävästi.

Käytettävyysvaatimuksen täyttäminen osoittautui haasteelliseksi. Työkalun toteutukseen oli käytetty jo kuukausia ennen kuin sille asetettiin käytettävyysvaatimus. Ennen vaatimusta työkalun käyttäjät olivat projektin sisältä, joka johti siihen, että käytettävyyteen ja sen visuaaliseen ilmeeseen käytettiin hyvin vähän aikaa. Sen sijaan toteutuksessa keskityttiin toiminnallisten vaatimuksien täyttämiseen.

## 6.2 Toteutusyksityiskohdat

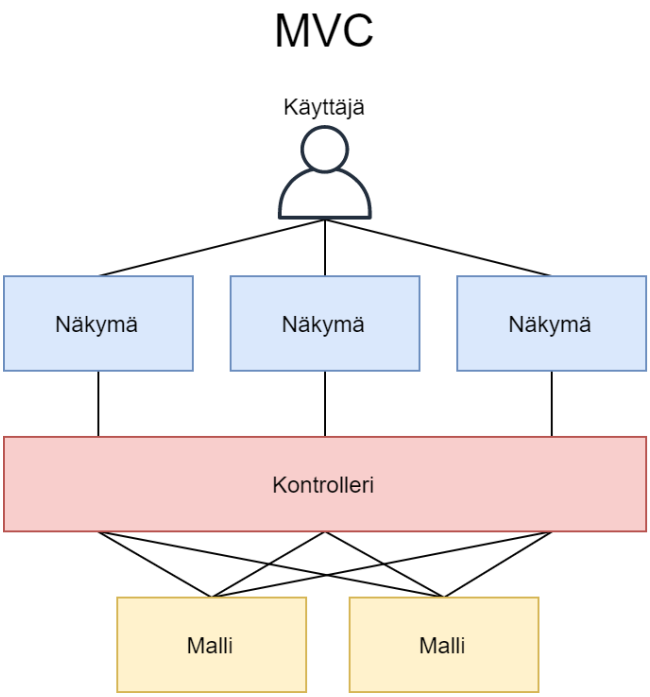
Vaikka mikropalveluarkkitehtuuri sekä erilliset front-end -sovellukset mahdollistavat eri teknologioiden valinnat mikropalveluiden ja työkalujen välillä, niin työn alussa päätettiin valita samat työkalut, kuin mitä muilla projektin osa-alueilla oli käytetty. Näin päätettiin siksi, että tulevaisuudessa muiden kehittäjien olisi helpompi lukea hallintatyökalun ohjelmakoodia, kun teknologiat ovat heille ennestään tuttuja. Tämän työn tekijälle Angular-sovelluskehys oli kokonaan uusi kokemus, kun taas back-endin .NET Core oli jokseenkin tuttu alusta. Lisäksi jos tulevaisuudessa työkaluun lisätään projektin laajuisia ominaisuuksia, kuten back-endissa keskitetty käyttäjienhallinta tai front-end siirretään osaksi jotain muuta kokonaisuutta, niin se hoituu tällöin helpommin.

Toteutuksessa olisi voitu hyödyntää enemmän reaaliaikaisuutta käytettävyyden parantamiseksi. Tässä toteutuksessa front-end saa datansa vasta kun se on kutsunut back-endin REST-rajapintaa ja back-end on suorittanut kaiken tarvittavan ja palauttanut vastauksen takaisin front-endille. Tämä ei ole tällaisessa tapauksessa tehokkain tai järkevin tapa välittää dataa back-endin ja front-endin välillä, koska hallintatyökalun tarkoituksena on vaikuttaa järjestelmän tilaan reaaliaikaisessa kontekstissa. Esimerkiksi toteutuksen alkuvaiheessa oli mahdollista, että järjestelmä tekee automaattisen tagimäärittelyn samaan aikaan, kun käyttäjä on tekemässä muutoksia siihen aiemmin back-endilta haettuun tagimäärittelyyn. Toinen samantyylinen tilanne on mahdollinen vielä toteutuksen

jälkeenkin. Siinä ottelun kokoonpanot ovat muuttuneet samaan aikaan, kun päivystäjä on tehnyt manuaalista tagimäärittelyä. Tässä tilanteessa hallintatyökalu ei siis vastaanota mitään tietoa tästä back-endissa tapahtuneesta kriittisestä muutoksesta.

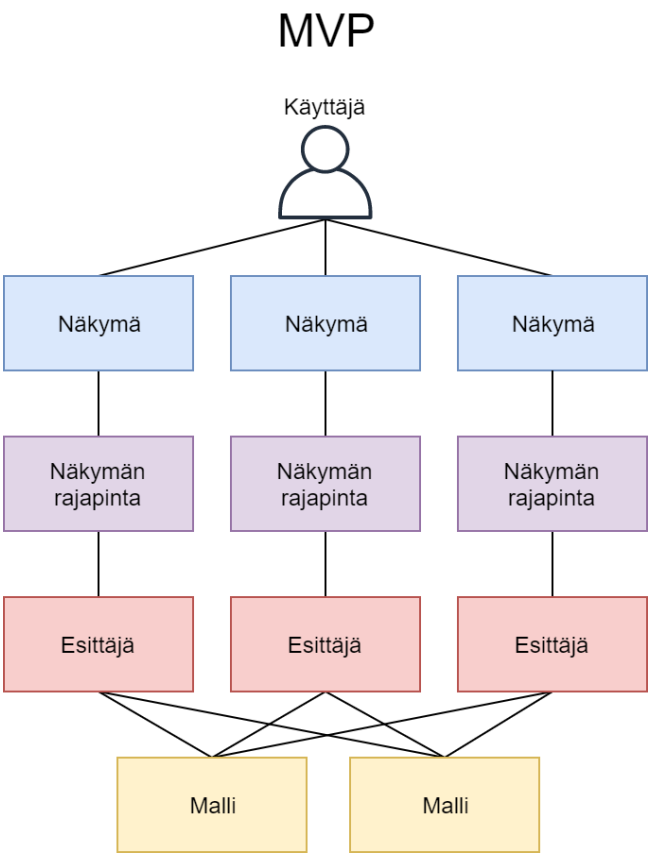
Reaaliaikaisuuden puutteesta johtuvat ongelmat voitaisiin käsitellä esimerkiksi hyödyntämällä WebSocket-protokollaa tai Server Sent Events (SSE) -teknologiaa. WebSocket on siis protokolla, joka mahdollistaa kaksisuuntaisen reaaliaikaisen kommunikaation palvelimen ja asiakkaan välillä. WebSocket-protokolla koostuu kahdesta osasta; *kättelyvaiheesta* (handshake) ja *datan siirtovaiheesta*. Kättelyvaiheen tarkoituksena on, että sekä asiakas että palvelin lähettävät erilliset kättelypyynnöt toisilleen, jolloin niiden välille luodaan TCP-yhteys, jos molempien kättelypyynnöt onnistuivat. Datan siirtovaiheessa asiakas ja palvelin voivat lähettää viestejä toisilleen samaisen TCP-yhteyden yli. Server Sent Events (SSE) on puolestaan tekniikka, jolla asiakassovellus voi vastaanottaa tapahtumia palvelimelta. Se on siis käytännössä asiakkaan päässä rajapinta, johon palvelin puskee tapahtumia HTTP-vastauksien muodossa. Siinä missä WebSocket-protokolla on tarkoitettu kaksisuuntaiseen kommunikaatioon, SSE on tarkoitettu yhdensuuntaiseen, palvelimelta asiakkaalle suunnattuun kommunikaatioon. Reaaliaikaisuutta ja .NET Core:n kirjastoja tutkimalla löydettiin SignalR-kirjasto [31], joka olisi sopinut hyvin tämän työn toteutukseen. Se on siis .NET-alustalle toteutettu kirjasto, joka helpottaa reaaliaikaisuuden toteuttamista .NET-ohjelmissa. SignalR olisi soveltunut tähän työhön erityisen hyvin, sillä se ei sitoudu käyttämään vain yhtä reaaliaikaista menetelmää, vaan sen avulla on mahdollista käyttää useita eri protokollia ja menetelmiä. Se automaattisesti vaihtaa protokollasta tai menetelmästä toiseen, jos sitä käyttävä .NET-sovellus tai asiakassovellus ei tue jotain protokollaa. [29, 30]

Hallintatyökalun front-end olisi voitu jakaa paremmin loogisiin komponentteihin. Tämän hetken toteutuksessa jokaiselle työkalulle on pääsääntöisesti yhdet komponentit, joiden vastuulla on hoitaa sekä komponenttiin liittyvä bisneslogiikka että datan visualisointi käyttäjälle. Komponentit olisi kannattanut suunnitella alusta alkaen siten, että ne noudattaisivat esimerkiksi MVP-arkkitehtuurimallia. Se tulee sanoista Model-View-Presenter ja onkin hyvin samankaltainen kuin back-endissa käytetty MVC, mutta MVP-mallia käytetään käyttöliittymätoteutuksissa. Siinä malli sekä näkymä ovat konseptiltaan samat kuin MVC:ssä, mutta MVP:n esittäjä (Presenter) tuo yhden abstraktiokerroksen näkymän ja bisneslogiikan välille. Alla olevissa kuvissa 14 ja 15 on demonstroitu MVC:n ja MVP:n eroa.




***Kuva 14.*** *MVC-mallin riippuvuudet.*

Yllä olevassa kuvassa 14 on kuvattu tyypillisen MVC-mallin abstraktiotasot. Hallintatyökalussa kuvan 14 näkymät tarkoittaisivat front-end -sovelluksen eri näkymien tekemiä HTTP-pyyntöjä back-endin yhteen tai useampaan kontrolleriin. Kontrollereita voi siis yksinkertaisuudessan kutsua moni eri näkymä.




***Kuva 15.*** *MVP-mallin riippuvuudet.*

Kuvassa 15 taasen on kuvattu MVP-mallin abstraktiota noudattava sovellus. Tässä tapauksessa kontrollerin sijasta käytetään esittäjiä, jotka ovat siis tiukasti sidottu tiettyyn näkymään. Esittäjän vastuulla on sitoa malli tai mallit siihen liitetyn näkymän rajapintaan. Tällä tavalla näkymät ovat täysin eristettyinä esittäjästä, jolloin komponenttilähtöisiä käyttöliittymäsovelluksia on helpompi testata ja toteuttaa. Hallintatyökalun front-end -sovellus olisi kannattanut toteuttaa MVP-mallina, sillä se sisältää useita eri työkaluja, joissa on useita eri komponenteiksi miellettäviä elementtejä.

## 6.3 Jatkokehitys

Vuoden 2019 alkupuoliskolla päätettiin, että hallintatyökalusta tehdään kokonaan uusi versio nykyisen hallintatyökalun rinnalle. Uuden hallintatyökalun tarkoituksena on, että se tarjoaisi kaikki samat ominaisuudet kuin tässä työssä toteutettu työkalu tarjoaa, mutta sen suunnittelussa otetaan huomioon käytettävyys heti projektin alusta alkaen. Tätä uutta työkalua tulisi käyttämään sekä Liigojen henkilöstö että projektin kehittäjät. Päätökseen vaikutti myös se, että hallintatyökalusta haluttiin pitää yksi toimiva versio koko ajan ajossa, joka sisältäisi juuri tässä työssä toteutetut ominaisuudet, eikä mitään muuta. Lisäksi uuteen hallintatyökaluun integroitaisiin uuteen järjestelmässä toteutettuun *autentikaatioon*, joka olisi johtanut suhteellisen laajoihin *refaktorointeihin* työssä toteutetussa

hallintatyökalussa, sillä kyseistä autentikointijärjestelmää ei ollut vielä tätä työkalua kehittäessä.

Autentikaation integrointi liittyy läheisesti uuteen suunniteltuun *päivystysmalliin*. Kaudella 2018-2019 päivystys oli projektin kehittäjien vastuulla, kun taas kaudelle 2019-2020 on tulossa kokonaan uusi hierarkkinen päivystymalli, joka myös itsessään vaikutti päätökseen toteuttaa uusi hallintatyökalu. Alustavan suunnitelman mukaan uusi päivystysmalli perustuu *päivystystasoihin*, joka käsittää ainakin kolme eri tasoa, joilla jokaisella on omat vastuualueensa. Matalimmalla tasolla operoi niin kutsuttu *hallipäivystäjä*, jonka tehtävänä on päivystää ottelua paikan päällä hallissa. Siitä korkeammalla tasolla operoi koko liigan laajuinen päivystäjä, jonka vastuulla on hallinnoida kaikkia kyseisen liigan otteluita. Nämä kaksi matalimman tason päivystäjää ovat liigan omaa henkilöstöä. Korkeimmalla tasolla operoi älykiekkoprojektin nimittämät päivystäjät, jotka operoisivat ikään kuin palvelupisteenä liigojen päivystäjille. Päivystystasot tuovat kokonaan uuden haasteen uudelle hallintatyökalulle, sillä hallintatyökalussa pitää myös sen myötä rajoittaa pääsyä resursseihin. Esimerkiksi SM-liigan päivystäjä ei saa tehdä tagimäärittelyä KHL:n otteluihin ja sama toisinpäin.

Tagien vääristymät ovat yksinkertaisesti listattu taulukkomuodossa nykyisessä toteutuksessa. Yksi jatkokehitysmahdollisuus olisi muodostaa näistä vääristymistä lämpökartta, joka kertoisi tagien viimeisimmän sijainnin. Tästä voitaisiin päätellä, onko mahdollisilla vääristymillä yhtäläisyyksiä sijainnin suhteen, jolloin myös voitaisiin eliminoida se mahdollisuus, että lokaattorit olisivat sijoitettu huonosti.

# 7. YHTEENVETO

Tämän työn tavoitteena oli kehittää web-pohjainen hallintatyökalu osaksi älykiekko-ottelun päivystysprosessia. Työkalun tarkoituksena oli helpottaa ja nopeuttaa ottelun päivystäjän toimintaa ennen ottelua sekä ottelun aikana. Lisäksi tarkoituksena oli tarjota mahdollisuus vaikuttaa järjestelmän tilaan ottelun aikana ja ottelun jälkeen. Tämä työ käsittää älykiekkojärjestelmän back-endiin toteutetun mikropalvelun sekä sitä käyttävän web-sovelluksen.

Reaaliaikaisesti analysoidun datan tuottamiseen ja esittämiseen sisältyy monia virhealttiita vaiheita. Työssä toteutettiin järjestelmään uusi mikropalvelu ja sitä varten web-käyttöliittymäsovellus, joka käytti kyseistä palvelua muun muassa tagimäärittelyn korjaukseen sekä muuhun järjestelmän operoimiseen. Hallintatyökalun vaatimuksena oli prosessin alussa ainoastaan tagimäärittelyn manuaalinen korjaaminen, mutta työkalun käyttöönoton myötä vaatimuksia asetettiin lisää sitä mukaa, kun koettiin tarpeelliseksi tuoda lisäominaisuuksia työkaluun. Tämän lisäksi aiemmin asetetut vaatimukset muuttuivat jatkuvasti järjestelmän kehittyessä.

Hallintatyökalu toteutettiin siten, että front-end -sovellus ja back-endin palvelu kommunikoivat keskenään HTTP-protokollan ylitse back-endiin toteutetun rajapinnan kautta. Front-end -sovellus toteutettiin komponenttiarkkitehtuuria noudattavaksi, sillä työkalulle asetetut vaatimukset voitiin jakaa loogisesti omiksi näkymikseen. Back-end -palvelu toteutettiin siten, että se jokseenkin noudattaa ASP.NET Core:n MVC-mallia, vaikka näkymä tässä tapauksessa tarkoittaa front-end -sovellusta.

Työkalun toteutusta arvioitiin sekä käytettävyyden että toiminnallisten vaatimusten näkökulmasta. Siinä käsiteltiin mahdollisuutta, että front-end -sovellus hakisi itse datansa suoraan mikropalveluarkkitehtuuria noudattavalta back-endilta. Tämä ratkaisu olisi ollut vain hieman tehokkaampi osassa tapauksista, sillä siinä olisi mahdollisesti säästetty ylimääräisiä RPC-kutsuja muihin palveluihin. Toisaalta tämä ei olisi ollut skaalautuva vaihtoehto ja siinä olisi sidottu front-end tiukasti back-endin palveluihin. Työssä toteutetulla tavalla mahdollistetaan hallintatyökalulle spesifin tiedon haku yhden rajapinnan kautta, jolloin front-end on löyhästi sidottu back-endiin. Käytettävyyden näkökulmasta front-end -sovellukseen jäi parantamisen varaa, sillä se ei hyödynnä reaaliaikaisia ratkaisuja.

Arvioinnin perusteella katsottiin, että tavoitteisiin päästiin osittain. Toiminnalliset vaatimukset toteutettiin, vaikka vaatimukset muuttuivat jatkuvasti. Hallintatyökalun käytettä-

vyyteen jäi parantamisen varaa. Jatkokehityksenä toteutetaan tämän työn hallintatyökalun rinnalle uusi työkalu, jossa panostetaan reaaliaikaisuuteen ja sitä kautta käytettävyyteen. Sen tarkoituksena on myös ratkaista uudistuneen päivystysmallin tuomat haasteet ja mahdolliset muut haasteet, joita tulevaisuudessa saattaa ilmentyä.

# LÄHTEET


[1] D. S. Mason, W. M. Foster, Putting moneyball on ice?, International Journal of Sport Finance, Vol. 2.4, 2007. https://www.researchgate.net/profile/William_Foster5/publication/5142759_Putting_Moneyball_on_Ice/links/55b65a3a08ae9289a08ac9d4/Putting-Moneyball-on-Ice.pdf

[2] Liiga. Saatavissa (viitattu 23.4.2019): https://liiga.fi/

[3] GET-Ligaen. Saatavissa (viitattu 23.4.2019): https://www.hockey.no

[4] KHL. Saatavissa (viitattu 23.4.2019): https://en.khl.ru

[5] Quuppa Positioning Engine. Saatavissa (viitattu 23.4.2019): https://quuppa.com/quuppa-positioning-engine/

[6] Quuppa. Saatavissa (viitattu 23.4.2019): https://quuppa.com

[7] Kafka. Saatavissa (viitattu 23.4.2019): https://kafka.apache.org

[8] Apache. Saatavissa (viitattu 24.4.2019): https://httpd.apache.org/

[9] C. Gomez, J. Oller, J. Paradells, Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Technology, Sensors, Vol. 12, 2012, s. 11734-11753

[10] R. Peng, M. L. Sichitiu. Angle of arrival localization for wireless sensor networks, 2006 3rd annual IEEE communications society on sensor and ad hoc communications and networks, Vol. 1, 2006, s. 374-382

[11] M. Glinz, On non-functional requirements, 15th IEEE International Requirements Engineering Conference, IEEE, 2007, s. 21-26

[12] Google Cloud Platform Overview, verkkosivu. Saatavissa (viitattu 23.4.2019): https://cloud.google.com/docs/overview/

[13] M. J. Kavis, Architecting the cloud: design decisions for cloud computing service models (SaaS, PaaS, and IaaS), John Wiley & Sons, 2014

[14] Docker, verkkosivu. Saatavissa (viitattu 28.4.2019): https://docs.docker.com/engine/docker-overview/

[15] Kubernetes, verkkosivu. Saatavissa (viitattu 28.4.2019): https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

[16] M. Luksa, Kubernetes in Action, Manning, vol. 1, 2018

[17] .NET Framework Guide, verkkosivu. Saatavissa (viitattu 29.4.2019): https://docs.microsoft.com/en-us/dotnet/opbuildpdf/toc.pdf

[18] Angular, verkkosivu. Saatavissa (viitattu 17.5.2019): https://angular.io

[19] TypeScript, verkkosivu. Saatavissa (viitattu 12.10.2019): https://www.typescript-lang.org/docs/home.html

[20] K. Stencel, P. Węgrzynowicz, Implementation variants of the singleton design pattern, OTM Confederated International Conferences “On the Move to Meaningful Internet Systems”, 2008, s. 396-406

[21] PostgreSQL. Saatavissa (viitattu 12.10.2019): https://www.postgresql.org

[22] Protocol Buffers. Saatavissa (viitattu 12.10.2019): https://developers.google.com/protocol-buffers

[23] gRPC. Saatavissa (viitattu 12.10.2019): https://grpc.io/docs/guides/

[24] HTTP – Hypertext Transfer Protocol. Saatavissa (viitattu 12.10.2019): https://www.w3.org/Protocols/

[25] RxJS. Saatavissa (viitattu 12.10.2019): https://rxjs-dev.firebaseapp.com/api

[26] P. Hamill, Unit Test Frameworks: Tools for High-Quality Software Development, O’Reilly, 2004

[27] Moq 4. Saatavissa (viitattu 5.10.2019): https://github.com/moq/moq4

[28] Linq. Saatavissa (viitattu 5.10.2019): https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/

[29] WebSocket. Saatavissa (viitattu 9.10.2019): https://tools.ietf.org/html/rfc6455

[30] Server-Sent Events. Saatavissa (viitattu 9.10.2019): https://www.w3.org/TR/eventsource/

[31] SignalR. Saatavissa (viitattu 7.10.2019): https://dotnet.microsoft.com/apps/aspnet/signalr